VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Rozšíření nástroje Visual Studio o podporu ladění nového programovacího jazyka

Extending Visual Studio to Support Debugging of a New Programming Language

2013 David Macek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **David Macek**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Rozšíření nástroje Visual Studio o podporu ladění nového programovacího jazyka
Extending Visual Studio to Support Debugging of a New Programming Language

Zásady pro vypracování:

V rámci dřívější diplomové práce byly prozkoumány některé možnosti, jak rozšířit vývojový nástroj Visual Studio 2010 o podporu nového programovacího jazyka. Jeden z úkolů, který v práci řešen nebyl je podpora ladění. Hlavním cílem bakalářské práce bude prozkoumat možnosti, jak lze realizovat ladění s využitím Visual Studia.

Cíle práce lze shrnout v těchto bodech:
1. Seznamte se s možnostmi integrace podpory dalšího programovacího jazyka do nástroje Visual Studio (více doporučená literatura).
2. Rozeberte možnosti, které vám Visual Studio nabízí.
3. Jako příklad vyberte vhodný jednoduchý jazyk a podporu ladění prakticky realizujte.

Seznam doporučené odborné literatury:

Lichovník Daniel: Podpora Vestavného procesně funkcionálního jazyka v nástroji Visual Studio, diplomová práce VŠB-TU Ostrava, 2010.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Marek Běhálek, Ph.D.**

Datum zadání: 18.11.2011

Datum odevzdání: 07.05.2013

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA Fakulta elektrotechniky a informatiky OSTRAVA 1

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7. května 2013 .....

Rád bych poděkoval Ing. Marku Běhálkovi, Ph.D. za pomoc při tvorbě práce.

Abstrakt

Tato bakalářská práce se zabývá problematikou rozšiřitelnosti vývojového prostředí Visual Studio o podporu ladění nového programovacího jazyka. První část rozebírá možnosti rozšíření, které Visual Studio poskytuje. Jsou zde popsány komponenty zajišťující podporu ladění na straně vývojového prostředí a to, jak jsou navzájem propojeny, a jsou naznačeny různé strategie propojení běhového prostředí s vývojovým prostředím v závislosti na zvoleném jazyku.

Druhá část podrobně popisuje praktická řešení a postupy potřebné pro tvorbu rozšiřujícího ladicího balíčku. Tyto poznatky jsou pak využity při implementaci podpory ladění jednoduchého jazyka, a to jak na straně Visual Studia, tak na straně běhového prostředí jazyka.

Klíčová slova: Visual Studio, ladění, rozšiřitelnost, DBGP

Abstract

This bachelor thesis concentrates on extensibility of the Visual Studio IDE regarding debugging support for a new programming language. First part is focused on extension points provided by Visual Studio, describing the components supporting debugging inside the IDE and their interconnections. This is followed by an overview of different strategies of connecting the language run-time with the IDE with regard to the choice of the language.

Detailed instructions and practical solutions for creating a debugging extension are given in the second part. This information is then used in the process of adding debugging support for a simple language to Visual Studio, as well as to the run-time of the language.

Keywords: Visual Studio, debugging, extensibility, DBGP

Obsah

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Psaní zdrojových kódů není jediná práce programátora při tvorbě a údržbě programů. Programátor musí také analyzovat a řešit chyby v programech. V rámci jedné iterace vývoje se můžou vyskytnout různé druhy chyb: chyby překladu, chyby běhu a logické chyby.

Nejdříve je kód při překladu (či v případě interpretovaných jazyků při explicitní syntaktické kontrole) zkontrolován, zda se v něm nevyskytují syntaktické chyby a porušení sémantiky jazyka. Tyto chyby překladu musí programátor všechny opravit, jinak nemůže pokračovat. Překladač navíc může provést další sémantické kontroly, zejména typovou kontrolu, nalezení nezachycených potenciálních výjimek, nalezení kódu čtoucího neinicializované proměnné a další, nicméně programátorovi jsou často buď výsledky nahlášeny jako pouhá varování a doporučení, anebo mu jsou dány prostředky, jak se těchto překážek zbavit bez toho, aby chyby opravil, například přetypování, zařazení typu výjimky mezi nekontrolované, inicializace proměnné nesmyslnou hodnotou...

Pokud se program podaří přeložit, může programátor zkusit program spustit a interagovat s ním. Při svém běhu se program může pokusit provést operaci, která není běhovým prostředím povolena. Pokud se tak stane a programátor v programu nezajistil ošetření takového stavu, je program běhovým prostředím ukončen. Ovšem ani když program dokončí svůj běh, nelze program prohlásit za prostý chyb. Program může vykazovat chování, které programátor nezamýšlel či neočekával, tedy obsahovat logickou chybu.

Poznámka 1.1 Hranice mezi třemi vyjmenovanými druhy chyb nejsou pevné. Definice jsou sice jednoznačné, ale stejný kus kódu se může v závislosti na kontextu objevit ve všech třech kategoriích. Například chybějící středník se obyčejně projeví jako chyba překladu, ovšem pokud středník chybí v kusu kódu, který je pak vyhodnocován konstrukcí typu eval, překlad proběhne v pořádku, ale program skončí běhovou chybou. Pokud je tento stav programem omylem zachycen a ignorován, projeví se jako logická chyba.

Pokud není příčina chyby běhu nebo logické chyby (často jsou souhrnně označovány slovem anglického původu bug) programátorovi okamžitě zřejmá, musí přistoupit k ladění. Ladění (anglicky debugging) programu je proces, při němž se programátor snaží najít příčinu chyby a odstranit ji. V závislosti na zkušenostech programátora a složitosti chyby může programátor použít k ladění různé metody. Běžně proces ladění začíná pokusem o opakovatelné znovunavození chyby, tedy nalezení postupu nebo vstupních dat, které spolehlivě chybu v programu vyvolají, přičemž se hledají vstupy co nejmenší a nejjednodušší. Následně je analyzován běh programu spuštěného s těmito vstupy, tedy zaznamenáván sled vykonávání jednotlivých rutin či příkazů v programu nebo vývoj vnitřního stavu programu. Takto získané informace pak programátor porovná s představou, jak by měl program fungovat, a na základě nalezených odlišností upraví kód.

Jelikož je ladění náročná a komplexní úloha, existuje poptávka po nástrojích, které ladění usnadňují. Podobně jako se vyvíjely nástroje podporující samotnou tvorbu programu – první programovací jazyky a jejich překladače, kompilátory, vysokoúrovňové jazyky, specializované editory a integrovaná vývojová prostředí – vyvíjely se i nástroje

podporující ladění – knihovny usnadňující tvorbu ladicích výpisů a záznamů, řádkové ladicí nástroje, grafické ladicí nástroje a integrovaná vývojová prostředí. Integrovaná vývojová prostředí (IDE) jsou nástroje, které umožňují v jednotném prostředí program psát, překládat, spouštět, ladit, nasazovat a zaznamenávat historii jeho vývoje. IDE jsou natolik dostupnými a nápomocnými nástroji, že si dnes snad nelze představit vývoj netriviálních programů bez jejich využití.

Jedním z populárních IDE pro platformu Windows je Visual Studio od společnosti Microsoft, kterému se věnuje i tato práce, a to problematice jeho rozšiřitelnosti v oblasti podpory ladění nového programovacího jazyka. Práce se zaměřuje na verzi 11.0, nazývanou Visual Studio 2012, která je v době psaní práce aktuální verzí. Toto IDE ve svém základním stavu poskytuje všechny funkce potřebné k ladění programů – propojení s běžícím programem, zobrazování jeho stavu v přímé návaznosti na zdrojový kód, sledování průběhu jeho vykonávání, ovládání jeho běhu (zastavování, posouvání po malých krocích), upravování jeho stavu a další. Mezi jazyky, které jsou plně podporovány v základní instalaci Visual Studia, patří C, C++ včetně několika rozšíření, VB.NET, C#, F# a JavaScript.

Kapitola 2 obsahuje úvod do obecné problematiky rozšiřitelnosti Visual Studia, na což navazuje kapitola 3 popisem základních konceptů a analýzou možných postupů při rozšíření Visual Studia o podporu ladění nového jazyka. Kapitola 4 jeden z uvedených postupů uvádí v podrobnostech a kapitola 5 pak popisuje využití tohoto postupu na praktické ukázce rozšíření Visual Studia o podporu ladění jazyka TL.

Poznámka 1.2 Součástí podpory ladění v některých edicích Visual Studia jsou i velmi pokročilé nástroje, jako IntelliTrace a Edit and Continue, které dále usnadňují a urychlují proces ladění. Tyto nástroje jsou mimo rozsah této práce a nebude jim dále věnována pozornost.

2 Visual Studio 2012 a jeho rozšiřitelnost

Prostředí Visual Studio (VS) vyvíjí společnost Microsoft od roku 1997 ve snaze sjednotit svá vývojová prostředí pro jednotlivé jazyky, Visual Basic, Visual C++ a Visual FoxPro. Samotné prostředí bylo psáno zejména v jazyku C++, ale od přibližně od roku 2001, spolu s uvedením platformy .NET, byly různé části přepisovány do jazyka C# a jednotlivé komponenty postupně oddělovány. Od VS 2005 po dnešní VS 2012 se prostředí vyznačuje stále větší modularitou a poskytuje větší možností rozšíření a zjednodušuje jejich implementaci.

2.1 Obecná architektura

To, co je uživatelem vnímáno jako jedno prostředí, jedna aplikace, je ve skutečnosti tvořeno desítkami komponent běžících nad frameworkem Visual Studio Shell. Komponenty jsou slučovány do balíčků zvaných VSPackages. Shell a komponenty poskytují každá jednu nebo více služeb a zároveň využívají služby poskytované ostatními[4]. Shell poskytuje tři základní služby:

SVsShell zajišťuje základní funkce a zabývá se registrací balíčků.

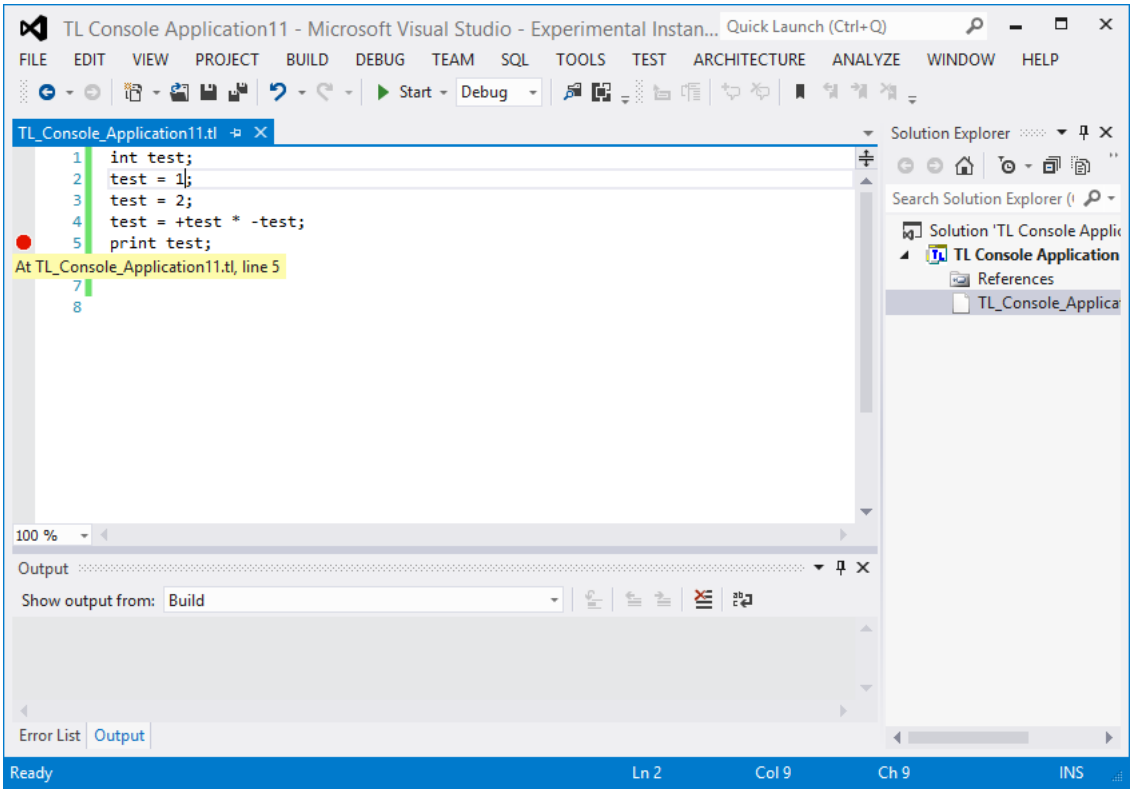
SVsSolution zajišťuje funkce související s řešeními (solutions) a projekty v nich obsaženými.

SVsUIShell zajišťuje uživatelské rozhraní – okna, panely, nabídky.

Všechny ostatní funkce dostupné v základní instalaci prostředí, včetně editorů kódu, designerů, podpory jednotlivých jazyků a podpory ladění, zajišťují balíčky. Služby navzájem komunikují pomocí rozhraní Component Object Model (COM) nebo pomocí Managed Extensibility Frameworku (MEF), přičemž určitá sada služeb je dostupná přes rozhraní COM (většinou architekturně starší části VS) a odlišná sada služeb je přístupná skrz MEF (zvláště části VS, které byly přepracovány ve verzi 2010).

COM, podobně jako například technologie CORBA či XPCOM, definuje rozhraní pro vzájemnou komunikaci objektů nezávislá na implementačním jazyku a platformě. Díky skutečnosti, že podpora COM je dostupná ve většině běžně užívaných jazyků, není rozšíření Visual Studia vázáno na znalost konkrétního jazyka. Drtivá většina balíčků je však psána v jazycích C++ a C#. Zejména při implementaci v jazyku C# (či jiných jazycích platformy .NET) je většina komplexity modelu COM před programátorem ukryta a ten se může soustředit na samotnou implementaci. Bohužel například registrace komponent COM i tak zůstává problematickým aspektem této technologie, ale pro tento konkrétní problém nabízí Visual Studio řešení, které bude popsáno v kapitole 4.

MEF, Visual Studiem prvně využitý ve verzi 2010, je framework umožňující rychlou tvorbu rozšiřitelných aplikací. MEF definuje metody propojení aplikace s rozšiřujícími komponentami za pomoci atributů definujících místa v kódu, která jsou otevřená k rozšíření (přijímají, importují funkcionalitu), a atributů definujících kusy kódu, které tvoří rozšíření (poskytují, exportují funkcionalitu). Jelikož jde o součást frameworku .NET



Obrázek 1: Visual Studio 2012 s otevřeným řešením a zdrojovým souborem

(prvně byl dodán s verzí 4.0), tvorba rozšíření využívajících MEF je efektivně omezená pouze na jazyky platformy .NET.

Pro vývoj rozšíření Visual Studia je společností Microsoft poskytována sada knihoven a nástrojů jménem Visual Studio SDK[17], dostupné bezplatně ke stažení[18]. Tyto nástroje zavedou do existující instalace Visual Studia nové druhy projektů (jako třeba již zmíněný VSPackage) a nastaví takzvanou experimentální instanci Visual Studia, tedy konfigurační profil oddělený od běžně používaného profilu, do kterého se instalují vyvíjená rozšíření. Testování a ladění rozšíření tak může probíhat se dvěma paralelně spuštěnými instancemi prostředí: v hlavní instanci se zobrazuje a případně upravuje kód rozšíření, v experimentální instanci se kód rozšíření vykonává, přičemž programátor může vykonávání kódu zastavovat, zkoumat vnitřní stav rozšíření a provádět všechny další ladicí úkony.

2.2 Architektura ladicích komponent

Funkcionalita ladění je ve Visual Studiu implementována základním balíčkem, který je napojený na další komponenty. Jsou to konkrétně následující součásti:

Debug package (UI) Ladicí balíček, poskytuje uživatelské rozhraní pro ladění a ukládá informace získané z ladicích jader pro další použití. Tato komponenta je součástí

Visual Studia.

Session debug manager (SDM) Správce ladicího sezení, komunikační komponenta, která předává informace mezi jednotlivými jádry a ladicím balíčkem. Pro ladicí balíček poskytuje jednotný proud událostí plynoucí z jednotlivých ladicích jader a rozděluje a přeposílá jim požadavky ladicího balíčku. Tato komponenta je součástí Visual Studia.

Process debug manager (PDM) Správce procesů a programů, udržuje seznam procesů a programů (tyto pojmy budou vysvětleny níže) dostupných k ladění. Tato komponenta je součástí Visual Studia.

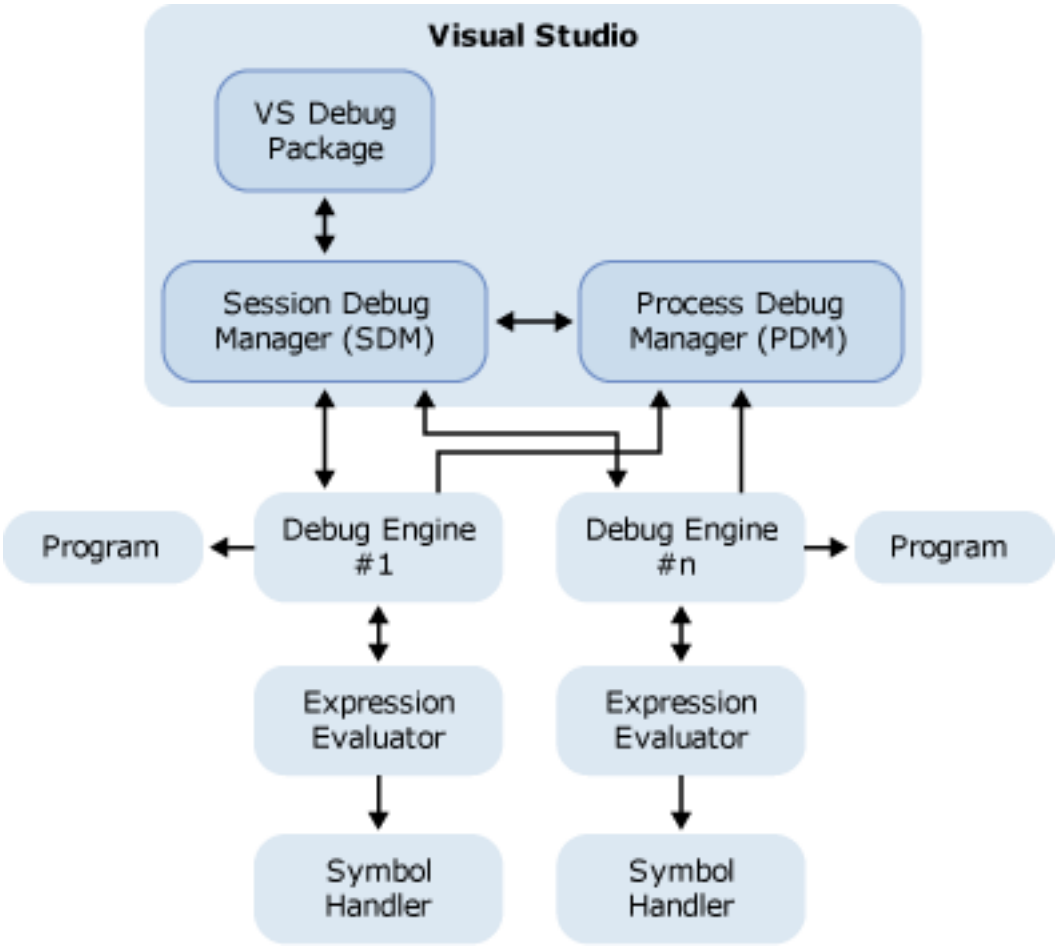
Debug engine (DE) Ladicí jádro, zajišťuje vykonávání ladicích požadavků ve spolupráci s běhovým prostředím, ve kterém program běží, což může být virtuální stroj či přímo operační systém. Pro každé běhové prostředí, ve kterém chceme ladit programy, je tedy potřeba zvláštní ladicí jádro. Jednoho ladicího sezení se může zúčastňovat více jader. Několik ladicích jader je součástí Visual Studia, řada dalších je dodávána třetími stranami.

Port supplier (PS) Poskytovatel portů, dodává správci procesů takzvané porty, jejichž účelem je vyjmenovat procesy běžící na stroji, umožnit spuštění a ukončování procesů a jejich připojení k ladicímu jádru. Visual Studio obsahuje port (s příslušným poskytovatelem) schopný pracovat s běžnými procesy na místním stroji a na vzdálených strojích s operačním systémem Windows, několik dalších portů a poskytovatelů je dodáváno třetími stranami.

Expression evaluator (EE) Vyhodnocovač výrazů, provádí na požádání syntaktickou analýzu výrazů a jejich vyhodnocování. Použitý vyhodnocovač je vybírán ladicím jádrem podle jazyka zdrojového souboru. Při vyhodnocování výrazů spolupracuje s poskytovatelem symbolů. Součástí Visual Studia jsou vyhodnocovače ke všem podporovaným jazykům.

Symbol handler (SH) Správce symbolů (někdy také symbol provider, SP), načítá z databází symbolů a z běhového prostředí seznam symbolů (proměnných, parametrů, metod, tříd), jejich význam a umístění v paměti. Součástí Visual Studia jsou správci symbolů, kteří jsou schopní číst databáze symbolů tvořené překladači dodávanými s Visual Studiem, a další jsou dodávány s ladicími jádry třetích stran. Poskytovatel symbolů nemusí být implementován jako zvláštní komponenta oddělená od ladicího jádra.

Type visualizer Vizualizátor dat, definuje způsob zobrazování hodnot určitého typu z programu. Vizualizátory spolupracují při získávání dat s vyhodnocovačem výrazů. Množství vizualizátorů je součástí Visual Studia, další jsou dodávány (stejně jako vyhodnocovače) s ladicími jádry třetích stran, ale jsou dostupné i vizualizátory v samostatných balíčcích, které spolupracují s některým z existujících ladicích jader.



Obrázek 2: Ladicí komponenty Visual Studia a vztahy mezi nimi

Propojení jednotlivých součástí je naznačeno v obrázku 2. Komponenty spravované Visual Studiem (komponenty UI, SDM a PDM) komunikují se zbylými komponentami pomocí řady rozhraní COM, která musí ladicí jádra, poskytovatelé portů, vyhodnocovače výrazů a správci symbolů implementovat. Vzhledem k tomuto faktu a výše uvedeným vlastnostem technologie COM byl pro tuto práci zvolen C# jako implementační jazyk.

2.3 Rozšiřitelnost ladění

Visual Studio poskytuje několik bodů rozšíření ladění:

Automatizace a makra Jedním z nejjednodušších způsobů rozšíření ladění je vytvoření makra využívajícího stávající schopnosti ladění novými způsoby. Například lze sestavit makro vypisující hodnoty lokálních proměnných a nechat ho spustit po každém zastavení programu. Pomocí maker však nelze dodat Visual Studiu nové schopnosti.

Vizualizátory Další jednoduchou metodou rozšíření je vylepšení způsobu zobrazování některých typů. Vizualizátor umožní nahlížet na data v programu novým, potenciálně přehlednějším nebo užitečnějším způsobem.

Vlastní EE Implementací vyhodnocovače výrazů lze Visual Studiu dodat schopnost analyzovat a vyhodnocovat výrazy v novém jazyce.

Vlastní SH Implementací správce symbolů lze do Visual Studia přidat podporu pro nové druhy databází symbolů.

Vlastní DE Implementace ladicího jádra zpřístupní Visual Studiu ladění programů v novém běhovém prostředí.

Vlastní port Implementace poskytovatele portů umožní Visual Studiu ladit procesy běžící v novém subsystému či na novém typu zařízení.

Tato práce se dále nevěnuje prvním dvěma uvedeným bodům, jelikož nepřináší žádné možnosti v oblasti, které se tato práce věnuje. V následující kapitole budou tyto možnosti podrobněji analyzovány se zaměřením na ty, které umožňují přidání podpory ladění programů v novém běhovém prostředí.

Poznámka 2.1 Vzhledem k nevelkému množství běžně používaných jazyků a běhových prostředí, jde o úlohu, se kterou se běžně programátoři nesetkávají. Navíc jde o komplexní úlohu, která v závislosti na žádané funkcionalitě může vyžadovat tvorbu či úpravu dalších netriviálních programů. Pravděpodobně z těchto důvodu je v době psaní práce k této problematice kromě oficiální dokumentace[5] dostupných pouze několik málo online zdrojů a téměř žádné publikace. Citace a odkazy v této práci proto pochází z online zdrojů.

3 Podpora ladění nového jazyka ve Visual Studiu

3.1 Terminologie

Jednou z prvních překážek při získávání informací o tomto tématu je poněkud nestandardní terminologie – některé koncepty získaly nejasné názvy a významy některých se dokonce liší od běžně používaných. Mezi hlavní objekty v ladicí architektuře Visual Studia patří:

Session manager Správce sezení, zajišťuje komunikaci IDE s ostatními komponentami. Správce vytváří dle potřeby potřebné objekty (ladicí jádro, server...), volá jejich metody a výsledky předává IDE. Zprávy správci jsou zasílány skrz rozhraní IDebugEventCallback2.

Session Ladicí sezení, množina všech procesů, které jsou laděny z jedné instance IDE.

Server Kontejner portů a poskytovatelů portů. Reprezentován rozhraním IDebugCoreServer2.

Port supplier Poskytovatel portů, dodává porty určitého typu. Reprezentován rozhraním IDebugPortSupplier2.

Port Reprezentuje spojení se zařízením nebo subsystémem a spravuje procesy na něm běžící. Dokáže procesy vyjmenovávat, spouštět, ukončovat a předávat je k ladění. Reprezentován rozhraním IDebugPort2, případně také IDebugPortEx2.

Engine Ladicí jádro, připojuje se k procesům a předává zpět další objekty, se kterými IDE během ladění komunikuje. Reprezentován rozhraním IDebugEngine2, případně také IDebugEngineLaunch2.

Program provider Poskytovatel programů, v procesech poskytnutých portem nalézá programy. Reprezentován rozhraním IDebugProgramProvider2.

Process Kontejner pro programy. Většinou odpovídá procesu operačního systému. Reprezentován rozhraním IDebugProcess2.

Program Nezávislá jednotka v procesu, která je předmětem ladění. Proces může obsahovat jeden nebo více programů – například jeden proces webového serveru může zároveň vykonávat několik nezávislých skriptů. Program zná svá vlákna a své moduly. Tento koncept se liší od významu stejnojmenného pojmu v teorii operačních systémů, kde program je množinou instrukcí a dat, který slouží jako šablona pro vytváření (spouštění) procesů. Reprezentován rozhraním IDebugProgram2, případně také IDebugProgram3 a IDebugEngineProgram2.

Program node Programový uzel, v některých kontextech zastupuje program a podává o něm základní informace. Reprezentován rozhraním IDebugProgramNode2.

Thread Vlákno, nezávislá jednotka vykonávání odpovídající vláknům operačního systému. Může být pozastaveno či běžet. Vlákno vykonává svůj proud instrukcí a je možné od něj zjistit seznam rámců na zásobníku (call stack), ve kterých se právě nachází. Reprezentován rozhraním IDebugThread2.

Stack frame Zásobníkový rámec, představuje kontext vykonávání, poskytuje informace o místních proměnných a parametrech. Ve většině programovacích jazyků vlákno vstupuje do nového rámce při zavolání funkce a při skončení funkce z rámce vystupuje. Reprezentován rozhraním IDebugStackFrame2.

Module Modul, samostatná jednotka kódu, například dynamicky načítaná knihovna (DLL, SO, Dylib). K modulům se načítají symboly. Reprezentován rozhraním IDebugModule2.

Breakpoint Místo v kódu, kde se má vykonávání zastavit. Breakpointy jsou reprezentovány v závislosti na kontextu a jejich stavu několika různými rozhraními:

IDebugPendingBreakpoint2 Čekající breakpoint. Do tohoto objektu jsou uloženy všechny potřebné informace při zadání breakpointu programátorem. Obsahuje kolekci navázaných a chybových breakpointů, které z tohoto breakpointu vznikly.

IDebugBoundBreakpoint2 Navázaný breakpoint. Instance tohoto rozhraní reprezentují konkrétní místa v proudu instrukcí, které odpovídají informacím z čekajícího breakpointu.

IDebugErrorBreakpoint2 Chybový breakpoint. Reprezentuje chybu při pokusu o vytvoření breakpointu, například špatně zadaný výraz u podmíněného breakpointu.

Memory context Místo v paměti. Reprezentován rozhraním IDebugMemoryContext2.

Code context Místo ve spustitelném kódu, tedy například adresu konkrétní instrukce stroje. Reprezentován rozhraním IDebugCodeContext2 a zároveň implementuje rozhraní IDebugMemoryContext2.

Document context Místo ve zdrojovém kódu. S jedním místem ve zdrojovém kódu může být asociováno více míst ve spustitelném kódu (například v případě instancí šablon jazyka C++). Reprezentován rozhraním IDebugDocumentContext2, je využíván spíše v komunikaci s IDebugCodeContext2.

Document position Taktéž odpovídá místu ve zdrojovém kódu. Reprezentován rozhraním IDebugDocumentPosition2, je využíván spíše v komunikaci s IDebugDocument2.

Document Dokument, představuje jeden soubor se zdrojovým kódem. Reprezentován rozhraním IDebugDocument2.

Expression evaluation context Vyhodnocovací kontext, představuje lexikální kontext pro vyhodnocování výrazů. Reprezentován rozhraním IDebugExpressionContext2.

Expression evaluator Provádí syntaktickou analýzu výrazů. Reprezentován rozhraním IDebugExpressionEvaluator.

3.2 Implementační strategie

Před implementací podpory ladění nového jazyka je potřeba zvolit správná místa rozšíření, se kterými se bude pracovat. Jak z předchozího textu může být patrné, volba nezávisí přímo na samotném jazyku, ale zejména na překladači jazyka a jeho výstupu a na běhovém prostředí.[6]

3.2.1 Nativní kód

Pokud kompilátor jazyka tvoří spustitelné soubory (.exe) ve strojovém kódu platformy, Visual Studio využije své zabudované ladicí jádro. Tato podpora je úplná, pokud jsou splněny podmínky, že kompilátor tvoří databáze symbolů ve formátu PDB a má syntax kompatibilní s C++.

V případě, že jazyk nemá syntax kompatibilní s C++, bude omezena možnost vyhodnocovat při ladění složené výrazy a zadávat podmínky k breakpointům. Zjevným řešením je implementovat vyhodnocovač výrazů, který zvolenému jazyku rozumí. V případě, že kompilátor tvoří databáze symbolů v jiném formátu než PDB, ladění bude omezeno na práci s proudem instrukcí bez jakékoli návaznosti na zdrojový kód. Zjevným řešením je implementovat správce symbolů, který rozumí formátu tvořenému zvoleným kompilátorem.

Velkou překážkou při implementaci těchto komponent je značný nedostatek jakékoli dokumentace. Oficiální texty sice (ač skoupě) popisují rozhraní těchto komponent, ale nikde nezmiňují možnost jejich oddělené implementace a způsob jejich registrace. Alternativním řešením druhého zmíněného nedostatku je doplnit schopnost tvorby PDB databází do kompilátoru nebo sestavit převaděč[21] z formátu podporovaného kompilátorem do formátu PDB, ovšem ani toto řešení není nijak snadné, protože formát PDB je uzavřený a není k dispozici oficiální specifikace. Další alternativa je zmíněna níže, v sekci 3.2.3.

Faktem je, že C a C++ jsou v době psaní práce až na výjimky jedinými jazyky kompilovanými do strojového kódu, v kterých jsou vyvíjeny programy ve Visual Studiu. Otázkou zůstává, jestli nedostatek dokumentace (a tím pádem nástrojů pro jiné jazyky) je příčinou, nebo důsledkem této jazykové dominance.

3.2.2 Řízený kód (.NET)

Jazyky implementované nad platformou .NET značně ulehčují práci implementátorům podpory ladění. Překladače jazyků platformy .NET tvoří standardní bytekód a databázi symbolů ve formátu PDB[7]. Visual Studio tak může použít svoje zabudované ladicí jádro a správce symbolů. Pro získání plné podpory ladění stačí implementovat vyhodnocovač[8], jehož nejnáročnější součástí je syntaktický analyzátor. V případě, že

syntaktický analyzátor pro jazyk je dostupný jako knihovna, jedná se o přímočarou integrační úlohu.

Vyhodnocovač je ve Visual Studiu reprezentován rozhraním IDebugExpressionEvaluator s třemi výkonnými metodami, zejména metodou Parse, která z řetězce vytvoří objekt, který libovolným způsobem reprezentuje analyzovaný výraz. Tento objekt, implementující rozhraní IDebugParsedExpression, pak metoda vrátí. Jediným požadavkem na tento objekt je metoda EvaluateSync, která za pomoci dodaných objektů (poskytujících například mapování symbolů na místa v paměti) výraz vyhodnotí a výsledek uloží do kontejneru s rozhraním IDebugProperty2.[9] Výhodou tohoto uspořádání je možnost opakovaného vyhodnocování výrazu bez nutnosti jej pokaždé znovu analyzovat.

3.2.3 Interpretovaný kód

V případě, že kompilátor jazyka netvoří ani nativní, ani řízený kód, je nutné přistoupit k implementaci ladicího jádra. Vlastní ladicí jádro je samozřejmě možné vytvořit i pro již podporovanou platformu[11], nicméně přínos takového řešení v poměru s jeho náročností je diskutabilní. Možnou výhodou takového přístupu by byla možnost připojit dokumentovaným způsobem k ladicímu jádru vlastní vyhodnocovač výrazů a vlastního správce symbolů a obejít tím potenciální omezení popsaná v sekci 3.2.1. Hlavní přínos implementace ladicího jádra však spočívá v možnosti ladit programy běžící v jiných běhových prostředích, zejména ve virtuálních strojích a interpretrech.

Vstupními body ladicího jádra jsou dvě třídy. Jedna implementující rozhraní IDebugEngine2 a IDebugEngineLaunch2 a druhá rozhraní IDebugProgramProvider2. Tyto dvě třídy odpovídají dvěma způsobům připojení ladicího jádra k programu. IDebugEngineLaunch2 slouží ke spuštění procesu přímo z IDE, zatímco IDebugProgramProvider2 je používán při připojení k již běžícímu procesu. Poskytovatel programů z běžících procesů (získaných z portu) vybírá ty, které obsahují programy podporované jádrem. Na základě výběru uživatele je pak na základě rozhraní IDebugEngine2 proces předán jádru k připojení.

Vzhledem k vzácnosti formátu PDB mimo nativní a .NET programy je navíc s velkou pravděpodobností nutné implementovat nějaký druh správce symbolů. Stejně tak pokud jde o nepodporovaný jazyk, je pro plnou podporu ladění potřebný i vyhodnocovač výrazů.

Jelikož tento způsob rozšíření podpory ladění je oproti ostatním dvěma uvedeným univerzálně použitelný a také komplexnější, bude se zbytek této práce věnovat právě jemu. V kapitole 4 jsou popsány podrobněji související koncepty a v kapitole 5 pak předvedeny části implementace na praktické ukázce.

3.2.4 Nestandardní procesy

Pokud jsou cílem ladění programy, které neběží ve standardních procesech (nedostupné přes výchozí port), je potřeba kromě ladicího jádra navíc implementovat port s rozhraními IDebugPort2 a IDebugPortEx2 a poskytovatele s rozhraním IDebugPortSupplier2, který bude porty vytvářet. Port pak na vyžádání vyjmenuje běžící procesy svého typu a umožňuje je spouštět a ukončovat. Objekty reprezentující proces implementují rozhraní IDebugProcess2

a IDebugProcessEx2 a programy získané z těchto procesů jsou implementacemi rozhraní IDebugProgram2 a IDebugProgramEx2.

Typickým příkladem situace, která vyžaduje tuto strategii, je potřeba ladit procesy běžící na jiném operačním systému, zejména na mobilních zařízeních nebo jejich emulátorech[19, 20]. Méně obvyklým využitím portů je ladění nestandardních lokálních procesů, ke kterým se Visual Studio nedokáže připojit[22, 23, 24].

Skutečnost, že je implementace portu uvedena jako strategie odlišná od implementace ladicího jádra, vychází z předpokladu, že pokud už pro cílovou platformu existuje implementace jádra s rozhraním analogickým k rozhraní pro Visual Studio, může jádro dodané s portem relativně jednoduše jenom informace přeposílat mezi správcem sezení a jádrem běžícím na cílovém systému.

4 Implementace ladicího jádra

Základem projektu ladicího jádra je, jako u jiných rozšíření, projekt typu VSPackage. Tento typ projektu se v nabídce objeví po instalaci balíku Visual Studio SDK a po potvrzení výběru tohoto typu nabídne krátkého průvodce, ve kterém je možno zvolit implementační jazyk (tato práce pracuje s jazykem C#), zadat základní informace o rozšíření a přiřadit podpisový klíč. Tento projekt bude pouze nosičem pro knihovnu s vlastním ladicím jádrem. Je tedy potřeba přidat do nově vzniklého řešení další projekt typu Class Library v jazyce C#. Tato knihovna bude poskytovat několik tříd zveřejněných do infrastruktury modelu COM a vícero dalších tříd, které není třeba zveřejňovat.

4.1 Instalace a registrace komponent

Výsledkem sestavení tohoto řešení je soubor typu VSIX, jehož spuštěním lze rozšíření nainstalovat do Visual Studia. V průběhu vývoje rozšíření jej však není potřeba nikam ručně instalovat, SDK instruuje Visual Studio k automatické instalaci rozšíření do experimentální instance po každé kompilaci. Aby bylo zajištěno, že knihovna s jádrem bude balena a instalována spolu s rozšířením, stačí otevřít soubor `source.extension.vsixmanifest` a do seznamu Assets přidat položku typu Assembly odkazující na druhý projekt.

Jelikož se jádro, poskytovatel programů a případné další zveřejněné komponenty musí před použitím zaregistrovat, je vhodné, aby tento proces proběhl automaticky při instalaci rozšíření. Jak již bylo zmíněno v kapitole 2, registrace komponent COM[12] zdaleka není přímočará bezstarostná záležitost. Typickým problémem je, že registrace komponent běžně probíhá v kontextu celého systému a proto vyžaduje administrátorská práva na místním počítači, což je však v rozporu se snahou izolovat vliv rozšíření na konkrétní instanci IDE. Visual Studio naštěstí poskytuje řešení obou problémů: pro své účely využívá kromě systémového (a uživatelského) katalogu komponent navíc katalog specifický pro danou instanci a při instalaci rozšíření do katalogu vloží všechny registrační informace uvedené v balíčku. Registrační informace vytváří při sestavení balíčku nástroj `RegPkg` spuštěním metod Register atributů Microsoft.VisualStudio.Shell.RegistrationAttribute deklarovaných na třídě reprezentující VSPackage (dědící z třídy Microsoft.VisualStudio.Shell.Package)[13].

4.2 Hostující proces

Využití technologie COM přináší nutnost dalšího rozhodnutí, a to zda jádro poběží v procesu běhového prostředí, nebo v procesu správce sezení (a potažmo celého IDE). Rozhodování může být ovlivněno několika faktory, zejména způsobem a množstvím komunikace s běhovým prostředím. Pokud je při vyhodnocování výrazů a vůbec načítání symbolů potřeba častého přenosu dat mezi běhovým prostředím a ladicím jádrem a pokud zároveň je možné, aby jádro komunikovalo s prostředím přímo, je vhodné pro zvýšení rychlosti přenosu dat jádro hostovat v procesu prostředí. Kvalita podpory technologie COM v jazyce, ve kterém je běhové prostředí napsáno, může být taktéž faktorem.[10]

Tato práce bude předpokládat hostování v procesu správce sezení. Díky abstrakcím poskytovaným modelem COM by však toto rozhodnutí nemělo mít na samotný proces implementace jádra velký vliv.

4.3 Kostra ladicího jádra

Funkční ladicí jádro musí implementovat desítky rozhraní, která definují celkem přes sto metod, přičemž větší část z nich může zůstat v první fázi neimplementována nebo je jejich implementace triviální. Tato práce proto nepopisuje postup jejich implementace a namísto toho počítá s využitím ukázkového jádra Debug Engine Sample (AD7Sample)[11]. Tento snad jediný ukázkový kód ladicího jádra je publikován pod otevřenou licencí MS-PL a stal se základem řady ladicích jader třetích stran[25, 26, 27, 28, 29, 30]. Kód AD7Sample obsahuje všechny potřebné definice, třídy a metody. Pokud z licenčních nebo jiných důvodů není možné použít tento kód, je možné implementace napsat znovu na základě dokumentace a komentářů v ukázkovém kódu.

Jelikož je kód AD7Sample funkčním jádrem, obsahuje části, které jsou pro toto jádro specifické a nebudou pravděpodobně prospěšné při implementaci nového jádra. Jedná se zejména o projekt `SampleEngineWorker` napsaný v C++, který zajišťuje komunikaci s ladicím API Win32. Jádro z projektu využívá zejména třídy DebuggedProcess, DebuggedModule a VariableInformation. Tyto třídy poskytují některá potřebná data reprezentující části laděného programu a je nutné je v pozměněné podobě reimplementovat nebo o relevantní části rozšířit jejich protějšky v jádru, tedy v pořadí: třídy AD7Engine, AD7Module a AD7Property. Využívány jsou ještě další třídy, Worker, ComponentException a Constants. První dvě jsou specifické pro ukázku a nejsou nadále potřeba a poslední zmiňovanou je možné nahradit třídou Microsoft.VisualStudio.VSConstants.

Dalším pomocným projektem je `ProjectLauncher`, který do Visual Studia přidává příkaz, kterým se dá spustit program v ladicím módu s připojeným jádrem AD7Sample. Výkonný kód tohoto projektu, nacházející se v metodě Connect.LaunchDebugTarget, je velmi krátký. Ideálním místem pro tento kód je metoda DebugLaunch rozhraní IVsDebuggableProjectCfg2 , které bývá implementováno jako součást podpory práce s projekty jazyka. Problematika rozšíření Visual Studia o podporu projektů nového jazyka byla řešena v dřívější práci[4] a nebude zde rozebírána.

Pro získání kostry ladicího jádra výše uvedeným způsobem je tedy potřeba:

1. Stáhnout a rozbalit balík AD7Sample.
2. Do projektu založeného pro nové ladicí jádro přesunout a zařadit soubory se zdrojovými kódy z projektu `SampleEngine`. Zároveň je potřeba přenést i odkazy na assemblies.
3. Doplnit do kódu pole či třídy pro data ukládaná ve výše zmíněné trojici tříd.
4. Nahradit reference na třídu Constants referencemi na třídu VSConstants. Odstranit reference na třídu ComponentException.

5. Všechny zbylé části kódu komunikující s třídami z projektu `SampleEngineWorker` smazat a označit (například komentářem TODO nebo vložením výjimky NotImplementedException. Reálné výstupní údaje nahradit za vhodné smyšlené údaje.

4.4 Zaregistrování komponent

Jednou z funkcí projektu `SampleEngineWorker` je i obstarání registrace do katalogu COM. Ta bude v novém projektu obstarána již zmíněnými atributy RegistrationAttribute. Konkrétně budou využity atributy ProvideDebugEngine a ProvideDebugLanguage, jejichž kód je zde uveden. Registrace v katalogu COM využívá identifikátoru zvaného CLSID, který je ve své podstatě identifikátorem typu GUID[14]. Tyto identifikátory jsou z definice globálně unikátní a proto je potřeba identifikátory použité v AD7Sample nahradit vlastními, například za pomoci nástroje `Tools > Create GUID` ve Visual Studiu. Celkem se jedná o čtyři identifikátory, po jednom CLSID pro třídy AD7Engine a AD7ProgramProvider, jeden GUID pro identifikaci jádra a jeden GUID pro identifikaci laděného jazyka. Výsledná hlavička deklarace pro VSPackage by pak mohla vypadat následovně (textový identifikátor jazyka musí být stejný jako identifikátor registrovaný rozšířením pro správu projektů v daném jazyce):

```
[ProvideDebugEngine("My Debugger", typeof(AD7ProgramProvider), typeof(AD7Engine),
    AD7Engine.Id)]
[ProvideDebugLanguage("MyLang", AD7Engine.LanguageId, AD7Engine.Id)]
[PackageRegistration(UseManagedResourcesOnly = true)]
[InstalledProductRegistration("#110", "#112", "1.0", IconResourceID = 400)]
[ComVisible(true)]
[Guid(GuidList.guidMyPackagePkgString)]
public sealed class MyProjectPackage : Microsoft.VisualStudio.Shell.Package {
```

Výpis 1: Použití registračních atributů

```
class ProvideDebugEngineAttribute : Microsoft.VisualStudio.Shell.RegistrationAttribute {
    private readonly string _id, _name;
    private readonly Type _programProvider, _debugEngine;

    public ProvideDebugEngineAttribute(string name, Type programProvider, Type debugEngine,
        string id) {
        _name = name;
        _programProvider = programProvider;
        _debugEngine = debugEngine;
        _id = id;
    }

    public override void Register(RegistrationContext context) {
        var engineKey = context.CreateKey("AD7Metrics\\Engine\\" + _id);
        engineKey.SetValue("Name", _name);
        engineKey.SetValue("CLSID", _debugEngine.GUID.ToString("B"));
        engineKey.SetValue("ProgramProvider", _programProvider.GUID.ToString("B"));
        engineKey.SetValue("PortSupplier", "{708C1ECA-FF48-11D2-904F-00C04FA302A1}");
        engineKey.SetValue("Attach", 1);
        engineKey.SetValue("AddressBP", 0);
```

```
            engineKey.SetValue("AutoSelectPriority", 6);
            engineKey.SetValue("CallstackBP", 1);
            engineKey.SetValue("Exceptions", 1);
            engineKey.SetValue("SetNextStatement", 1);
            engineKey.SetValue("RemoteDebugging", 1);
            engineKey.SetValue("HitCountBP", 0);
            engineKey.SetValue("EngineClass", _debugEngine.FullName);
            engineKey.SetValue("EngineAssembly", _debugEngine.Assembly.FullName);
            engineKey.SetValue("LoadProgramProviderUnderWOW64", 1);
            engineKey.SetValue("AlwaysLoadProgramProviderLocal", 1);
            engineKey.SetValue("LoadUnderWOW64", 1);

            using (var incompatKey = engineKey.CreateSubkey("IncompatibleList")) {
                incompatKey.SetValue("guidCOMPlusNativeEng", "{92EF0900-2251-11D2-B72E-0000
                    F87572EF}");
                incompatKey.SetValue("guidCOMPlusOnlyEng", "{449EC4CC-30D2-4032-9256-
                    EE18EB41B62B}");
                incompatKey.SetValue("guidScriptEng", "{F200A7E7-DEA5-11D0-B854-00
                    A0244A1DE2}");
            }
            using (var autoSelectIncompatKey = engineKey.CreateSubkey("AutoSelectIncompatibleList"))
                {
                autoSelectIncompatKey.SetValue("guidNativeOnlyEng", "{3B476D35-A401-11D2-AAD4
                    -00C04F990171}");
            }

            var clsidKey = context.CreateKey("CLSID");
            var clsidGuidKey = clsidKey.CreateSubkey(_debugEngine.GUID.ToString("B"));
            clsidGuidKey.SetValue("Assembly", _debugEngine.Assembly.FullName);
            clsidGuidKey.SetValue("Class", _debugEngine.FullName);
            clsidGuidKey.SetValue("InprocServer32", context.InprocServerPath);
            clsidGuidKey.SetValue("CodeBase", Path.Combine(context.ComponentPath, _debugEngine.
                Module.Name));
            clsidGuidKey.SetValue("ThreadingModel", "Free");
            clsidGuidKey = clsidKey.CreateSubkey(_programProvider.GUID.ToString("B"));
            clsidGuidKey.SetValue("Assembly", _programProvider.Assembly.FullName);
            clsidGuidKey.SetValue("Class", _programProvider.FullName);
            clsidGuidKey.SetValue("InprocServer32", context.InprocServerPath);
            clsidGuidKey.SetValue("CodeBase", Path.Combine(context.ComponentPath, _debugEngine.
                Module.Name));
            clsidGuidKey.SetValue("ThreadingModel", "Free");

            using (var exceptionAssistantKey = context.CreateKey("ExceptionAssistant\\KnownEngines
                \\" + _id)) {
                exceptionAssistantKey.SetValue("", _name);
            }
        }

        public override void Unregister(RegistrationContext context) {
        }
    }
```

Výpis 2: Třída atributu ProvideDebugEngine

```
class ProvideDebugLanguageAttribute : Microsoft.VisualStudio.Shell.RegistrationAttribute {
    private readonly string _guid, _languageName, _engineGuid;

    public ProvideDebugLanguageAttribute(string languageName, string guid, string engineGuid)
        {
        _languageName = languageName;
        _guid = guid;
        _engineGuid = engineGuid;
    }

    public override void Register(RegistrationContext context) {
        var langSvcKey = context.CreateKey("Languages\\Language Services\\" +
            _languageName + "\\Debugger Languages\\" + _guid);
        langSvcKey.SetValue("", _languageName);
        var eeKey = context.CreateKey("AD7Metrics\\ExpressionEvaluator\\" + _guid + "\\{994
            B45C4-E6E9-11D2-903F-00C04FA302A1}\\Engine");
        eeKey.SetValue("Language", _languageName);
        eeKey.SetValue("Name", _languageName);
        eeKey.SetValue("Engine", _engineGuid);
    }

    public override void Unregister(RegistrationContext context) {
    }
}
```

Výpis 3: Třída atributu ProvideDebugLanguage

4.5 Ladicí logika

Nyní by mělo být rozšíření ve stavu, kdy je ho možné sestavit a nainstalovat a je možné ladicí jádro z IDE vyvolat spuštěním projektu, nicméně zatím bez efektu. V tuto chvíli je možné do dříve označených míst (popsáno v sekci 4.3) začít doplňovat výkonný kód specifický pro zvolené běhové prostředí. Pokud není komunikace s prostředím navržena podle nějakého standardu, pro který existuje komunikační knihovna pro .NET, je na místě zároveň připravit třídu či třídy, které budou komunikaci obsluhovat. Důraz při volbě či tvorbě komunikačního rozhraní a jeho implementace je nutné klást na umožnění asynchronní komunikace. Správce sezení totiž od ladicího jádra očekává, že bude na události v laděném programu obratem reagovat zasláním objektu s rozhraním IDebugEvent2 metodě Event objektu rozhraní IDebugEventCallback2. Objekt tohoto rozhraní je jádru předán zároveň s požadavkem na připojení k procesu, ať už běžícímu nebo novému.

Jednoduchou variantou, jak postupovat s tvorbou výkonného kódu jádra, je začít s rudimentární podporou životního cyklu procesu a následně ji zdokonalovat. Programátor pak bude mít možnost se v každé iteraci dopracovat k spustitelnému řešení a příslušnou funkcionalitu řádně otestovat. Následující postup je jednou z vhodných možností:

1. spouštění a připojování procesů spolu se zasíláním informací o normálním skončení procesu
2. odpojování a ukončování procesů

3. nastavování breakpointů spolu se zasíláním informací o pozastavení vykonávání
4. obnovení vykonávání
5. předávání skutečných informací o modulech a zásobníkových rámcích
6. krokování
7. další...

4.5.1 Životní cyklus procesu

4.5.1.1 Vytvoření a připojení Jádro může být k procesu připojeno třemi způsoby:

1. Ladicí jádro je požádáno o vytvoření procesu metodou IDebugEngineLaunch2.LaunchSuspended. Nově vytvořený pozastavený proces je následně předán metodě IDebugEngineLaunch2.ResumeProcess, který skrz port oznámí existenci programů v tomto procesu. Správce sezení na tuto událost zareaguje předáním programu metodě IDebugEngine2.Attach. Ladicí jádro je povinno ohlásit události IDebugEngineCreateEvent2 (pouze jednou pro danou instanci jádra), IDebugProgramCreateEvent2 a IDebugLoadCompleteEvent2.
2. Obdobným způsobem může vytvoření procesu obstarat port pomocí metod IDebugPortEx2.LaunchSuspended a IDebugPortEx2.ResumeProcess.
3. Proces byl spuštěn dříve, nezávisle na IDE. Poskytovatel programů je pomocí metody IDebugProgramProvider2.GetProviderProcessData požádán o programové uzly reprezentující programy laditelné jádrem. Uživatelem vybrané programy jsou pak předány metodě IDebugEngine2.Attach.

Který způsob správce zvolí, závisí na tom, zda jde o spuštění nového procesu či připojení k běžícímu, zda bylo při spuštění požádáno (například na základě konfigurace projektu) o konkrétní ladicí jádro a který z objektů implementuje příslušné nepovinné rozhraní s příponou Ex2.

Obecné implementace metod LaunchSuspended, ResumeProcess a Attach jádra dodržujících popsaná pravidla mohou vypadat následovně:

```
int IDebugEngineLaunch2.LaunchSuspended(string pszServer, IDebugPort2 port, string exe,
    string args, string dir, string env, string options, enum_LAUNCH_FLAGS launchFlags, uint
    hStdInput, uint hStdOutput, uint hStdError, IDebugEventCallback2 ad7Callback, out
    IDebugProcess2 process) {
  this.Callback = ad7Callback;

  ProcessStartInfo info = new ProcessStartInfo(exe, args) { WorkingDirectory = dir };
  Process proc = Process.Start(info);
  this.Program = new AD7Program(proc);

  AD_PROCESS_ID adProcessId = new AD_PROCESS_ID();
  adProcessId.ProcessIdType = (uint)enum_AD_PROCESS_ID.AD_PROCESS_ID_SYSTEM;
  adProcessId.dwProcessId = (uint)Program.ProcessId;
  port.GetProcess(adProcessId, out process);
```

```
    return VSConstants.S_OK;
}
```

Výpis 4: Metoda IDebugEngineLaunch2.LaunchSuspended

```
int IDebugEngineLaunch2.ResumeProcess(IDebugProcess2 process) {
    IDebugPort2 port;
    process.GetPort(out port);
    IDebugDefaultPort2 defaultPort = (IDebugDefaultPort2)port;
    IDebugPortNotify2 portNotify;
    defaultPort.GetPortNotify(out portNotify);

    portNotify.AddProgramNode(new AD7ProgramNode(process));
    // expect Attach to be called

    return VSConstants.S_OK;
}
```

Výpis 5: Metoda IDebugEngineEx2.ResumeProcess

```
int IDebugEngine2.Attach(IDebugProgram2[] rgpPrograms, IDebugProgramNode2[]
      rgpProgramNodes, uint celtPrograms, IDebugEventCallback2 ad7Callback,
      enum_ATTACH_REASON dwReason) {
    if (this.Program == null)
    {
        // running program
        this.Callback = ad7Callback;
        this.Program = new AD7Program(tgpProgram[2]);
    }
    else
    {
        // program from LaunchSuspended
    }

    AD7EngineCreateEvent.Send(this);
    AD7ProgramCreateEvent.Send(this);

    // actually connect to the runtime
    this.Program.Connect();
    // send IDebugLoadCompleteEvent2 later

    return VSConstants.S_OK;
}
```

Výpis 6: Metoda IDebugEngine2.Attach

4.5.1.2 Nastavení breakpointů Po dokončení inicializace by mělo být jádro spojeno s běhovým prostředím a program by měl být připraven k běhu. Ještě předtím však správce požádá o nastavení všech breakpointů, které uživatel zadal. K tomuto účelu jádro implementuje metodu IDebugEngine2.CreatePendingBreakpoint přijímající parametr IDebugBreakpointRequest2 a vracející instanci IDebugPendingBreakpoint2, která obsahuje

informace o vzniklém čekajícím breakpointu. Tento objekt je pak požádán pomocí metody Bind o navázání. V tuto chvíli, potenciálně za účasti správce symbolů, by měly být breakpointy skutečně vloženy do programu. IDE může s rozhraním IDebugPendingBreakpoint2 dále pracovat, zejména vyžádat seznam navázaných a chybových breakpointů, vypnout či zapnout breakpoint a nastavovat výraz s podmínkou pro zastavení. O každém novém navázaném breakpointu by mělo být IDE informováno zasláním události IDebugBreakpointBoundEvent2, o chybě zasláním události IDebugBreakpointErrorEvent2. Ladicí jádro má taktéž za úkol se pokusit navázat breakpointy při načtení nového modulu.

4.5.1.3 Spuštění a zastavení Jakmile IDE příslušně nastavilo všechny parametry a breakpointy, požádá o spuštění vykonávání programu a očekává příchod událostí. Důležitými událostmi, které musí být vyvolány aspoň jednou při startu běhu programu, jsou IDebugModuleLoadEvent2 a IDebugThreadCreateEvent2, značící v pořadí: nahrání modulu a vznik nového vlákna. Opačné události, tedy uvolnění modulu a zánik vlákna, jsou reprezentovány rozhraními IDebugModuleLoadEvent2 a IDebugThreadDestroyEvent2. (Mezi událostmi nahrání a uvolnění modulu IDE může rozlišit zavoláním metody GetModule.) Běh programu může být přerušen v několika případech:

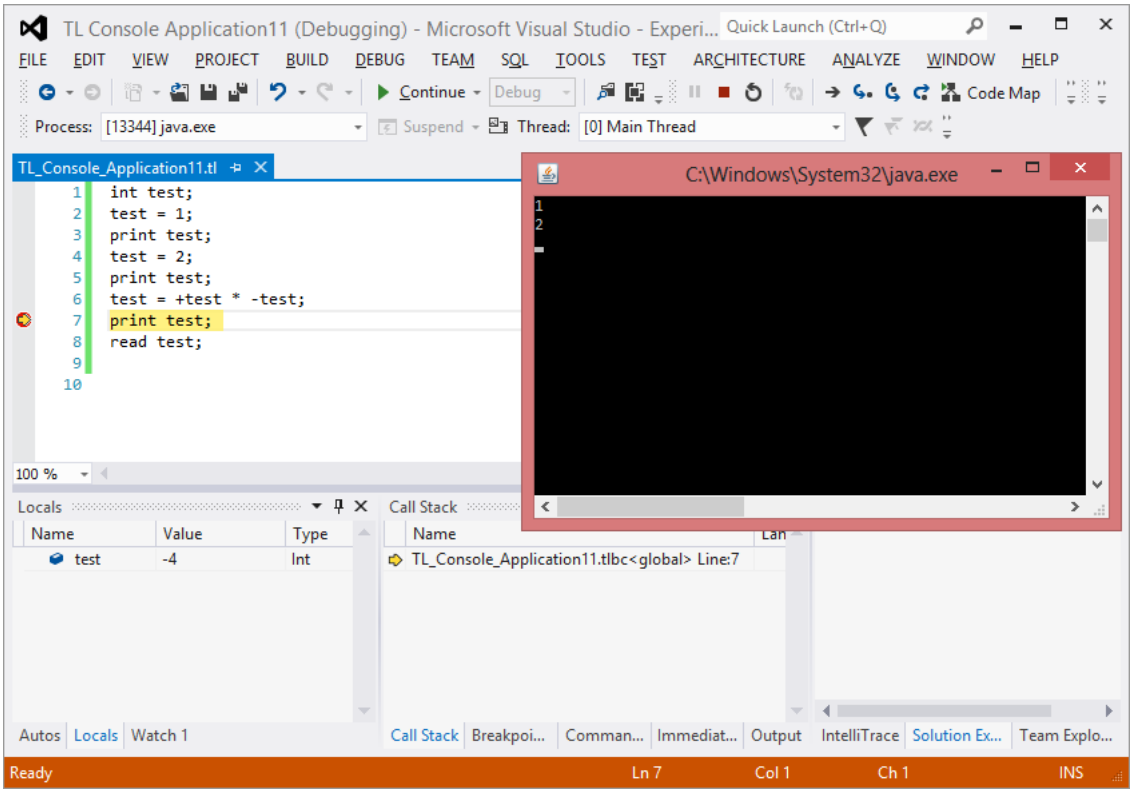
- Program dokončil vykonávání kroku. Správce sezení je o tomto informován příchodem události IDebugStepCompleteEvent2.
- Program se zastavil na breakpointu. Správci je zaslána událost IDebugBreakpointEvent2.
- V programu vznikla chyba. Odpovídajícím rozhraním pro příslušnou událost je IDebugExceptionEvent2.
- Uživatel požádal o přerušení běhu programu. Správce sezení na základě tohoto požadavku zavolá metodu CauseBreak. Metodu tohoto názvu definují tři rozhraní: IDebugProcess2, IDebugProgram2 (a metoda IDebugEngineProgram2.Stop) a IDebugEngine2. Požadovanou akcí je ve všech případech přerušení běhu, nicméně každé rozhraní má odlišnou oblast vlivu, konkrétně v pořadí: přerušení běhu všech programů v konkrétním procesu, konkrétního programu a všech programů laděných konkrétním jádrem. O úspěšném přerušení běhu programu je správce informován příchodem události IDebugBreakEvent2.

4.5.1.4 Získávání údajů Pokud je program přerušen, IDE zobrazí uživateli informace o aktuálním stavu programu (obrázek 3) a poskytne mu možnosti dalšího ovládání programu. Na pokyn uživatele může IDE žádat o přidávání, mazání a modifikaci breakpointů, obnovení běhu programu, změnu dat programu a další úkony. Mezi standardně zobrazovanými informacemi je seznam modulů, seznam vláken, zásobníky volání, hodnoty a typy lokálních proměnných; další informace může uživatel získat na požádání.

Seznam modulů a vláken by IDE mohlo teoreticky tvořit z jednotlivých objektů zasílaných s událostmi IDebugModuleLoadEvent2 a IDebugThreadCreateEvent2, nicméně (například v situaci po připojení k běžícímu procesu) je nutné, aby program uměl poskytnout seznamy na vyžádání (metody IDebugProgram2.EnumThreads a IDebugProgram2.EnumModules. Od

modulů (rozhraní IDebugModule2) je možné zjistit pouze několik informací. Objekty představující vlákna (rozhraní IDebugThread2) musí být schopné poskytnout zejména aktuální seznam zásobníkových rámců (metoda IDebugThread2.EnumFrameInfo). Po rámcích (rozhraní IDebugStackFrame2) je zase vyžadován seznam lokálních proměnných, parametrů, umístění aktuální instrukce v paměti, umístění ve zdrojovém kódu atd. Oficiální dokumentace rozhraní a komentáře v kódu AD7Sample jsou v tomto ohledu dostatečnými podklady pro dokončení výkonného kódu.

Poznámka 4.1 U implementací, které si žádané informace pro zvýšení výkonu ukládají, by mělo být dbáno na bezvadnost schopnosti rozeznat, kdy je nutné z běhového prostředí načíst čerstvé informace. V opačném případě hrozí, že IDE bude uživateli zobrazovat zastaralé údaje. Stejně tak by měla být sledována přesnost údajů – ukázalo se, že pokud se IDE dozví, že k zastavení došlo na jiném místě v kódu, než kde bylo původně nastaveno, IDE automaticky požádá o obnovení běhu programu.



Obrázek 3: Informace zobrazované v režimu pozastaveného ladění

4.5.1.5 Obnovení běhu

Pro obnovení běhu dává jádro k dispozici tři základní metody a dvě rozšířené. IDE preferuje metody rozhraní IDebugProcess3 a IDebugProgram3, po-

kud jsou k dispozici. Nabídku těchto možností, jak je prezentována uživateli IDE, ukazuje obrázek 4.

Step Vykonat jeden krok (podle předaných parametrů může být krok představován řádkem, příkazem či vstupem do funkce). Po dokončení vykonání kroku Metoda je definována na rozhraních IDebugProcess3 a IDebugProgram2.

Continue Pokračovat ve vykonávání stávajícím způsobem. Pokud k současnému zastavení došlo v průběhu krokování, následující zastavení by mělo nastat po vykonání stanoveného kroku. Metoda je definována na rozhraních IDebugProcess3 a IDebugProgram2.

Execute Pokračovat ve vykonávání běžným způsobem. Pokud k současnému zastavení došlo v průběhu krokování, je toto nastavení zrušeno. Metoda je definována na rozhraních IDebugProcess3 a IDebugProgram2.

ExecuteOnThread Pokračovat ve vykonávání konkrétního vlákna určeným způsobem. Metodě je předáno jako parametr vlákno a také způsob pokračování, který nabývá hodnot odpovídajících třem uvedeným způsobům obnovení běhu – step, continue, execute. Metoda je definována na rozhraní IDebugProgram3.

Resume Pokračovat ve vykonávání konkrétního vlákna. Tato metoda má význam zejména při jednotlivém zacházení s vlákny a je v blízkém vztahu s metodou Suspend. Obě metody jsou definovány na rozhraní IDebugThread2.

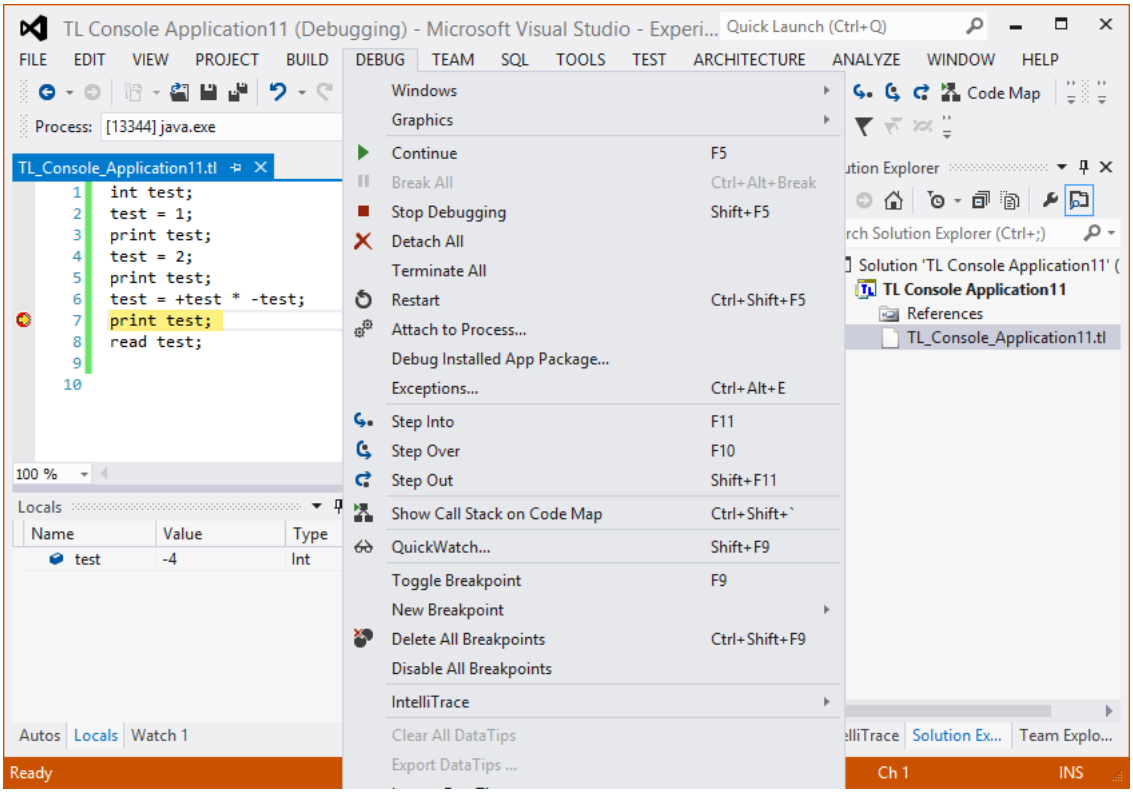
4.5.1.6 Skončení

Ke skončení ladicího sezení může dojít buď na vyžádání, nebo po úspěšném dokončení běhu programu:

- Pro násilné ukončení běhu programu jsou definovány metody IDebugProgram2.Terminate , IDebugProcess2.Terminate, IDebugEngineLaunch2.TerminateProcess, IDebugPortEx2.TerminateProcess . První uvedená metoda ukončuje běh programu, zbylé tři ukončují celý proces. Volba metody je uskutečněna na základě způsobu spuštění či připojení k programu.
- Pro odpojení jádra od běžícího programu je k dispozici metoda IDebugProgram2.Detach. Po odpojení od jádra program pokračuje ve svém vykonávání dále nepřerušen.
- Po vykonání poslední instrukce program úspěšně dokončil svůj běh. Správce sezení je o tomto informován událostí IDebugProgramDestroyEvent2.

4.6 Události

V této kapitole bylo zmíněno množství událostí, které ladicí jádro posílá správci sezení. Zaslání je uskutečněno předáním objektu události metodě IDebugEventCallback2.Event spolu s jejím GUID a dalšími informacemi o původu události. Objekt IDebugEventCallback2 je jádru předán při spuštění či připojení k programu a jádro si musí na něj uložit referenci pro pozdější užití. Při zaslání události jsou předávány atributy určující způsob reakce správce sezení:



Obrázek 4: Nabídka ladicích možností v režimu pozastaveného ladění

Asynchronous Na událost není třeba nijak reagovat.

Synchronous Ladicí jádro očekává synchronní odpověď na událost zavoláním metody IDebugEngine2.ContinueFromSynchronousEvent.

Stopping Vykonávání programu bylo pozastaveno.

Sync Stop Vykonávání programu bylo pozastaveno a ladicí jádro očekává synchronní odpověď zavoláním některé z metod Execute, Step nebo Continue rozhraní IDebugProgram2.

Expression Evaluation Předmětem události je vyhodnocený výraz.

Mezi události, které ještě nebyly zmíněny, patří:

IDebugSymbolSearchEvent2 Jádro nahrálo k modulu symbolické informace.

IDebugOutputStringEvent2 Program zaslal ladicí hlášku.

IDebugErrorEvent2 Nastala chyba, o které má být informován uživatel.

IDebugMessageEvent2 Nastal stav, o kterém má být informován uživatel, a je od něj očekáváno rozhodnutí.

4.7 Shrnutí

V této kapitole byly detailně popsány koncepty a situace, se kterými se programátor setká při tvorbě ladicího jádra pro nové běhové prostředí, a poskytnuty návody pro vytvoření potřebných projektů a pro implementaci nejdůležitějších částí jádra. Záměrně nebyla probrána pokročilejší témata, protože jejich vyčerpávající popis by výrazně překročil rozsah této práce. Byly taktéž vynechány konkrétní ukázky výkonného kódu, protože ten je často specifický pro zvolené laděné prostředí a nebyl by příliš nápomocen v řešení podpory jiných prostředí. Výjimkou z tohoto tvrzení může být řešení využívající rozšířených standardů k získání podpory pro více běhových prostředí naráz nebo alespoň k značnému zvýšení znovupoužitelnosti výkonného kódu. V následující kapitole bude představeno praktická ukázka právě takového řešení, implementovaná jako součást této práce.

5 Implementace podpory ladění jazyka TL ve Visual Studiu

Při tvorbě implementace, která je součástí této práce, byla následována všechna rozhodnutí a všechny postupy uvedené v předchozích kapitolách. Jedná se tedy o holé ladicí jádro doplněné o základní podporu projektů, projekt je napsán v jazyce C#, k instalaci do Visual Studia je využit VSPackage a hostujícím procesem jádra je proces IDE.

Pro uvedení čtenáře do kontextu je první část této kapitoly věnována problematice samotného jazyka a běhového prostředí. V druhé části jsou popsány vlastnosti jazyka a prostředí přímo ovlivňující ladění. V třetí části jsou pak popsány vybrané problémy řešené při tvorbě ladicího jádra pro Visual Studio.

5.1 Jazyk TL

Jazyk, který byl zvolen pro tuto praktickou ukázku, vznikl jako seminární projekt do bakalářského předmětu o programovacích jazycích a překladačích [1]. Jazyk byl nazván Test Language (TL) a jde o jednoduchý strukturovaný jazyk se syntaxí algolovského typu, jehož gramatika patří do třídy LL(1). V TL nelze definovat žádné znovupoužitelné jednotky kódu jako funkce či moduly; TL ani neposkytuje žádné knihovní či zabudované funkce kromě zápisu na standardní výstup a čtení ze standardního vstupu. Podrobné specifikace jazyka a jeho bytekódu se nacházejí v příloze B.

Jednoduchost jazyka nijak nesnižuje názornost praktické ukázky – architektura jádra totiž umožňuje bezproblémové rozšíření o podporu funkcí, modulů a vláken.

5.1.1 Překlad

Překladač jazyka je napsaný v Javě a pro syntaktickou analýzu využívá nástroj JavaCC[38]. Díky jeho použití nepotřebuje překladač speciální lexikální analyzátor a syntaktický analyzátor je automaticky generován z popisu gramatiky. V průběhu analýzy, prováděné metodou shora dolů, probíhá převod do syntaktického stromu, o něž se starají kousky kódu doplněné do předlohy analyzátoru. Po získání syntaktického stromu provede překladač typovou kontrolu, a pokud nenajde žádné chyby, vygeneruje rekurzivním sestupem syntaktickým stromem bytekód pro zásobníkový stroj s pamětí s náhodným přístupem.

5.1.2 Popis běhového prostředí

Běhové prostředí je taktéž napsáno v Javě a po svém spuštění načte celý soubor s bytekódem do seznamu instrukcí v paměti a jednou ho zběžně projde. V tomto průchodu ověří, zda se jedná o bytekód podporované verze a zapíše si umístění návěstí. Následně začne instrukce jednu po druhé vykonávat.

5.2 Podpora ladění v TL

5.2.1 Symbolické informace

Jelikož ve své původní verzi překladač a běhové prostředí neobsahovaly žádnou podporu ladění, bylo nutné nějakou navrhnout a do těchto nástrojů doplnit. První překážkou ladění byl fakt, že běhové prostředí nebylo schopné nijak přiřadit instrukce k jednotlivým příkazům ve zdrojovém kódu ani přiřadit místa v paměti jednotlivým proměnným. Jedním možným řešením by bylo nahradit překladač a zásobníkový stroj interpretrem. Schůdnějším řešením však bylo upravit specifikaci bytekódu. Bytekód byl pouze doplněn o nevýkonné instrukce `var` a `line`, takže byla zachována zpětná kompatibilita (dopředná kompatibilita řešení nebyla žádána).

5.2.2 Ovládání virtuálního stroje

V původní verzi virtuální stroj vykonával instrukce jednu za druhou a nebylo možné jej nijak ovládat. Rozhraní stroje bylo tedy rozšířeno o podporu asynchronního ovládání běhu, správy breakpointů a získávání informací o stavu stroje. Zároveň bylo definováno rozhraní pro ladicí rozšíření, kterému je předáno ovládací rozhraní stroje a jsou mu oznamovány události pocházející ze stroje.

5.2.3 Komunikační protokol

Jakmile bylo připraveno vnitřní ladicí rozhraní, bylo potřeba rozhodnout, jakým způsobem jej zveřejnit. Jako velmi flexibilní řešení se nabízí komunikace po síti pomocí TCP spojení. Před návrhem vlastního komunikačního protokolu byla dána přednost použití existujícího rozšířeného protokolu. Takovým protokolem je DBGP[3], hybridní textový a XML protokol s množstvím existujících implementací[31, 32, 33, 34, 35, 36, 37]. Tento protokol má taktéž širokou podporu ze strany vývojářských prostředí.

Příkazy běhovému prostředí mají formát vycházející ze syntaxí řádkových nástrojů: jednoslovné jméno příkazu následované páry slov vyjadřujícími vždy název parametru a jeho hodnotu. Všechny příkazy jsou ukončovány symbolem `NUL` (znak s číselným kódem 0). Odpovědi běhového prostředí jsou ve formátu XML včetně deklarací, přičemž každá zpráva je uvozena dekadicky zapsanou délkou těla zprávy v bytech, symbolem `NUL`; tímto symbolem je taky ukončena. Důvody pro tento rozdíl mezi formátem příkazů a odpovědí jsou několikeré: formát příkazů ulehčuje jejich ruční zápis, zatímco formát odpovědí umožňuje přirozeně vyjadřovat strukturovaná data. Navíc takto běhové prostředí nemusí obsahovat žádnou XML knihovnu – ke tvorbě validního XML totiž často stačí jednoduché spojování řetězců.

Protokol má další vlastnosti, umožňující např. vypořádat se s příchodem zpráv mimo pořadí, omezovat délku zpráv a dynamicky zjišťovat stav podpory rozšířených příkazů. Podrobná specifikace protokolu se nachází v elektronické příloze (popis v příloze A). Výpisy 7 a 8 ukazují příklady komunikace.

```
run -i tlde15 [NUL]
```

```
158 [NUL]
<?xml version='1.0' encoding='utf-8' ?><response xmlns='urn:debugger_protocol_v1'
    transaction_id='tlde15' command='run' status='break' reason='ok'></response> [NUL]
```

Výpis 7: Ukázka příkazu protokolu DBGP a odpovědi na něj

```
<init appid='0' thread='1' idekey='tldebug' language='TL' protocol_version='1.0' fileuri =' file: ///
    C:/Program/program.tlbc'></init>

breakpoint_set -i tlde1 -t line -f " file: /// C:/Program/program.tlbc" -n 15
breakpoint_set -i tlde2 -t line -f " file: /// C:/Program/program.tlbc" -n 5
breakpoint_set -i tlde3 -t line -f " file: /// C:/Program/program.tlbc" -n 9
context_get -i tlde4
stack_get -i tlde5

<response transaction_id='tlde1' command='breakpoint_set' id='15'></response>
<response transaction_id='tlde2' command='breakpoint_set' id='5'></response>
<response transaction_id='tlde3' command='breakpoint_set' id='9'></response>
<response transaction_id='tlde4' command='context_get'>
   <property name='b' type='Int'><![CDATA[0]]></property>
   <property name='a' type='Int'><![CDATA[0]]></property>
</response>
<response transaction_id='tlde5' command='stack_get'>
   <stack level='0' type=' file ' filename=' file: /// C:/Program/program.tlbc' lineno='1' />
</response>

run -i tlde15

<response transaction_id='tlde15' command='run' status='break' reason='ok'></response>

stack_get -i tlde16
context_get -i tlde17

<response transaction_id='tlde16' command='stack_get'>
   <stack level='0' type=' file ' filename=' file: /// C:/Program/program.tlbc' lineno='15' />
</response>
<response transaction_id='tlde17' command='context_get'>
   <property name='b' type='Int'><![CDATA[0]]></property>
   <property name='a' type='Int'><![CDATA[1]]></property>
</response>

breakpoint_remove -i tlde18 -d 15

<response transaction_id='tlde18' command='breakpoint_remove'></response>

run -i tlde44

<response transaction_id='tlde44' command='run' status='stopping' reason='ok'></response>
```

Výpis 8: Výňatek z komunikace v průběhu ladicího sezení

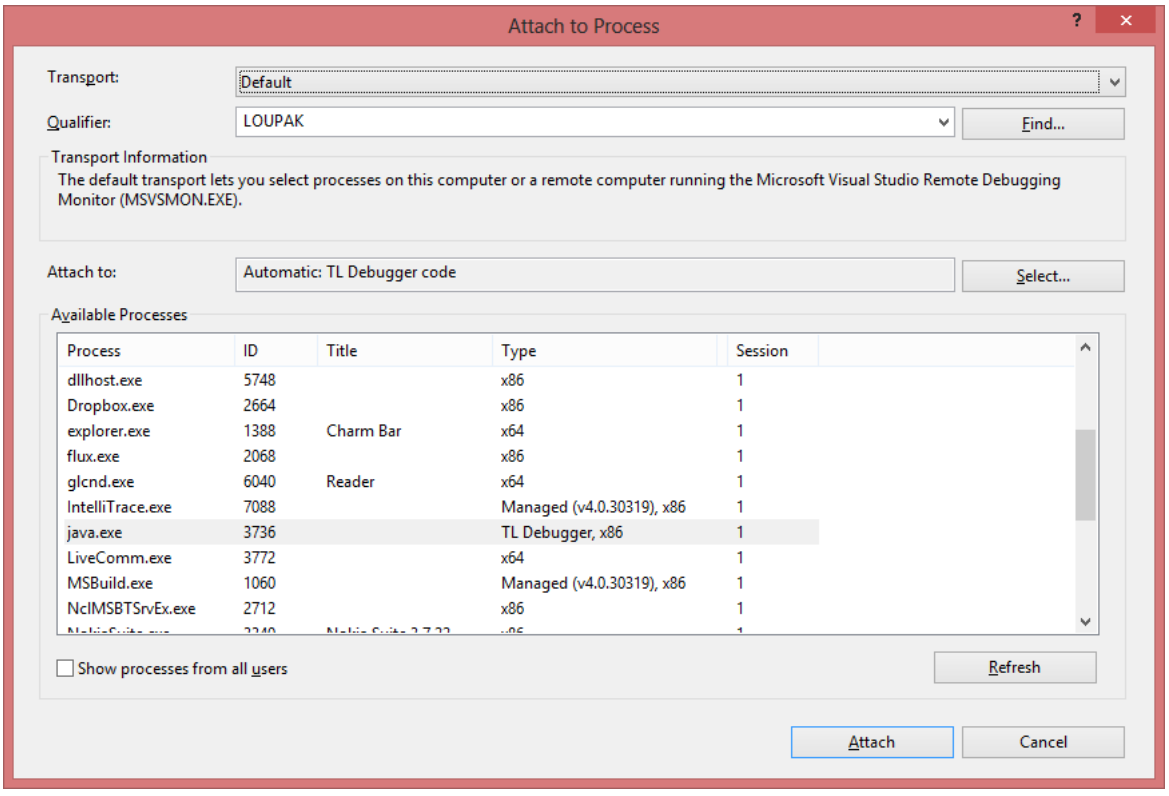
Komunikace na straně běhového prostředí probíhá asynchronně, ve zvláštních vláknech, aby na její uskutečnění prostředí nemuselo čekat. Zároveň je tak možné přijmout

příkaz v průběhu vykonávání, například příkaz pro pozastavení běhu na vyžádání. Implementace v běhovém prostředí podporuje podmnožinu příkazů protokolu a až na několik zjednodušení plně odpovídá specifikaci.

5.3 Ladicí jádro

5.3.1 Připojení k běžícímu procesu

Poskytovatel programů tohoto jádra filtruje z nabízených procesů pouze procesy `java`, jelikož běhové prostředí samo je vykonáváno virtuálním strojem Javy. Protokol DBGP neumožnuje vyžádat připojení za běhu programu ze strany ladicího nástroje, proto je potřeba, aby bylo běhové prostředí spuštěno v ladicím režimu se správnými parametry – běhové prostředí pak před započetím vykonávání programu čeká, až bude ladicí jádro připraveno na navázání spojení. Na obrázku reffig:VSDebugAttach lze vidět, jak Visual Studio prezentuje informace získané z poskytovatelů jednotlivých jader uživateli a jak na základě informací uvedených při registraci demonstračního jádra automaticky volí kombinaci jader, která se budou ladění účastnit.



Obrázek 5: Okno pro výběr procesu a jádra k připojení

5.3.2 Asynchronní komunikace

Na straně ladicího jádra jsou smyčky pro příjem a odesílání zpráv taktéž vykonávány ve vlastních vláknech. Tato asynchronnost však vyžaduje zvláštní událostí. Například pokud je okamžitě po zastavení programu o tomto vyvolána událost, IDE začne žádat informace o zásobníku volání a o lokálních proměnných. Z návrhu protokolu a faktu, že jádro nemá přímý přístup do virtuálního stroje, ovšem vyplývá, že o tyto informace je nutné nejdříve požádat přes síť a počkat na odpověď, což je pomalé. Namísto toho jádro, jakmile se dozví o zastavení, vyšle obratem požadavky na zjištění těchto informací a až po jejich přijetí předá událost dále. Malými úpravami by bylo možné dosáhnout toho, aby byla událost předána také okamžitě a informace mohly dorazit, zatímco IDE událost zpracovává. K neužitečnému čekání by pak došlo, pouze pokud by IDE stihlo zažádat o zásobník volání ještě před příchodem odpovědí z druhé strany ladicího spojení.

Bylo taktéž potřeba vyřešit nevhodné načasování požadavků na nastavení breakpointů. IDE o jejich nastavení žádá v brzké fázi spouštění programu a není zaručeno, že je v danou chvíli už navázáno a inicializováno spojení s běhovým prostředím. Breakpointy si proto jádro musí pamatovat a žádat o jejich nastavení, až je spojení skutečně připraveno. Metoda pro příjem zpráv vypadá přibližně takto:

```
private void MessageReceived(object sender, MessageReceivedEventArgs eargs) {
    /* ... */
    switch (rootElementName) {
        case "init":
            /* ... */
            m_engine.Callback.OnModuleLoad(Module);
            m_engine.Callback.OnThreadStart(Thread);

            Connection.StartSending();
            foreach (uint addr in pendingBreakpoints)
                AddBreakpoint(/*...*/);
            pendingBreakpoints.Clear();

            onStackAndContextReceived = () => {
                Run();
            };
            GetContext();
            GetStack();
            break;
        case "response":
            /* ... */
            switch (command) {
                case "context_get":
                    /* ... */
                    if (onStackAndContextReceived != null)
                        gotContext = true;
                    break;
                case "stack_get":
                    /* ... */
                    if (onStackAndContextReceived != null)
                        gotStack = true;
                    break;
```

```
                case "break":
                    onStackAndContextReceived = () => {
                        m_engine.Callback.OnAsyncBreakComplete(Thread);
                    };
                    GetStack();
                    GetContext();
                    break;
                }
                break;
        }
        if (gotStack && gotContext && onStackAndContextReceived != null)
        {
            gotStack = false;
            gotContext = false;
            Thread.StackFrame = new AD7StackFrame(/*...*/);

            onStackAndContextReceived();
            onStackAndContextReceived = null;
        }
    }
```

Výpis 9: Schéma implementace metody pro příjem zpráv od běhového prostředí

6 Závěr

Jak bylo zmíněno v předchozím textu, rozšíření Visual Studia o podporu ladění nového jazyka není úlohou snadnou ani podrobně zdokumentovanou. Tato práce byla stvořena s cílem poskytnout další zdroj informací pro zájemce o ujmutí se této úlohy. Byly popsány různé přístupy k rozšíření a cíle, jichž je možno jednotlivými přístupy dosáhnout, a byly podrobně rozebrány koncepty a postupy potřebné pro implementaci ladicího jádra, jedné ze složitějších alternativ.

Implementace, která je výsledkem této práce, umožňuje provádět všechny základní úkony ladění, tedy spustit program v režimu ladění, ovládat vykonávání programu a zkoumat jeho stav v plné návaznosti na zdrojový kód programu, v jednotném a známém rozhraní, které Visual Studio poskytuje. Implementace samozřejmě nevyužívá všech nabízených možností a může se podle postupů naznačených v předchozích kapitolách dále vyvíjet. Další vývoj komponenty by mohl směřovat k přidání podpory krokování, vyhodnocování složitých výrazů či podpory pro změnu stavu programu za běhu.

Fakt, že implementace je prezentována na nepoužívaném jazyku, samozřejmě snižuje rozsah okamžitých využití, nicméně ladicí jádro je na tomto jazyku nezávislé a díky použití protokolu DBGP je možné toto řešení adaptovat na jiné, populární jazyky, jejichž běhová prostředí podporují tento standardní protokol. Taková adaptace by kromě několika triviálních úprav sestávala zejména z rozšíření ladicího jádra o podporu vlastností onoho nového jazyka, které jazyk TL nepodporuje.

7 Reference

[1] BĚHÁLEK, Marek. *Zadání semestrálního projektu předmětu PJP* [online]. [cit. 2013-04-01].
<http://www.cs.vsb.cz/behalek/vyuka/pjp/projekt2/index.php>.

[2] BĚHÁLEK, Marek. *Specifikace jazyka pro semestrální projekt předmětu PJP* [online]. [cit. 2013-04-01].
<http://www.cs.vsb.cz/behalek/vyuka/pjp/projekt2/popis.php>.

[3] CARAVEO, Shane a Derick RETHANS. *DBGP – a common debugger protocol for languages and debugger UI communication* [online]. revize 17 [cit. 2013-04-01].
<http://xdebug.org/docs-dbgp.php>.

[4] LICHOVNÍK, Daniel. *Podpora vestavného procesně funkcionálního jazyka v nástroji Visual Studio*. Diplomová práce na Fakultě elektrotechniky a informatiky VŠB-TU Ostrava, 2010. Vedoucí diplomové práce Ing. Marek Běhálek, Ph.D.

[5] *Visual Studio Debugger Extensibility* [online]. [cit. 2013-04-01].
<http://msdn.microsoft.com/en-us/library/vstudio/bb161718.aspx>.

[6] *Roadmap for Extending the Debugger* [online]. [cit. 2013-04-01].
<http://msdn.microsoft.com/en-us/library/bb145316.aspx>.

[7] *Common Language Runtime and Expression Evaluation* [online]. [cit. 2013-04-01].
<http://msdn.microsoft.com/en-us/library/bb161369.aspx>.

[8] *Writing a Common Language Runtime Expression Evaluator* [online]. [cit. 2013-04-01].
<http://msdn.microsoft.com/en-us/library/bb161694.aspx>.

[9] *Expression Evaluator Architecture* [online]. [cit. 2013-04-01].
<http://msdn.microsoft.com/en-us/library/bb161301.aspx>.

[10] *Choosing a Debug Engine Implementation Strategy* [online]. [cit. 2013-04-01].
<http://msdn.microsoft.com/en-us/library/bb147022.aspx>.

[11] *Visual Studio Debug Engine Sample* [online]. [cit. 2013-04-01].
<http://archive.msdn.microsoft.com/debugenginesample>.

[12] *Registering COM Applications* [online]. [cit. 2013-04-01].
<http://msdn.microsoft.com/en-us/library/windows/desktop/ms683954.aspx>.

[13] *How to: Register a VSPackage* [online]. [cit. 2013-04-01].
<http://msdn.microsoft.com/en-us/library/bb166544.aspx>.

[14] *CLSID Key* [online]. [cit. 2013-04-01].
<http://msdn.microsoft.com/en-us/library/ms691424.aspx>.

[15] *Properties.Item Method* [online]. [cit. 2013-04-01].
<http://msdn.microsoft.com/en-us/library/envdte.properties.item.aspx>.

[16] *Exceptions debugging CustomProject Sample* [online]. [cit. 2013-04-01].
<http://mpfproj10.codeplex.com/discussions/273104>.

[17] *Visual Studio SDK* [online]. [cit. 2013-04-01].
<http://msdn.microsoft.com/en-us/library/vstudio/bb166441.aspx>.

[18] *Microsoft Visual Studio 2012 SDK* [online]. [cit. 2013-04-01].
<http://www.microsoft.com/en-us/download/details.aspx?id=30668>.

[19] *Xamarin.Android* [online]. [cit. 2013-04-01].
<http://xamarin.com/monoforandroid>.

[20] *Carbide.vs Overview* [online]. [cit. 2013-04-01].
<http://www.developer.nokia.com/Community/Wiki/Archived:Carbide.vs_Overview>.

[21] *cv2pdb – converter of DMD CodeView and GDC DWARF debug information to PDB debug format* [online]. [cit. 2013-04-01].
<http://dsource.org/projects/cv2pdb>.

[22] *VisualDDK – develop/debug Windows drivers directly from Visual Studio* [online]. [cit. 2013-04-01].
<http://visualddk.sysprogs.org/>.

[23] *Debugging Interix processes with gdb and WinDbg* [online]. [cit. 2013-04-01].
<http://www.suacommunity.com/faqs.aspx#401>.

[24] *WinGDB – debugging with GDB under Visual Studio* [online]. [cit. 2013-04-01].
<http://www.wingdb.com/>.

[25] *Node.js Debugging Support for Visual Studio* [online]. [cit. 2013-04-01].
<https://github.com/omgtehlion/NodeVsDebugger/>.

[26] *PowerGUI Visual Studio Extension* [online]. [cit. 2013-04-01].
<http://powerguivsx.codeplex.com/>.

[27] *Python Tools for Visual Studio* [online]. [cit. 2013-04-01].
<http://pytools.codeplex.com/>.

[28] *MagoWrapper – a .NET library for mago internals that provides generic D program debug support* [online]. [cit. 2013-04-01].
<https://github.com/aBothe/MagoWrapper>.

[29] *MySQL Connector/Net* [online]. [cit. 2013-04-01].
<http://dev.mysql.com/doc/refman/5.6/en/connector-net.html>.

[30] *C# Open Source Managed Operating System* [online]. [cit. 2013-04-01].
<http://cosmos.codeplex.com/>.

[31] *AutoHotkey Debugging Features* [online]. [cit. 2013-04-01].
<http://l.autohotkey.net/docs/AHKL_DBGPClients.htm>.

[32] *Xdebug extension for PHP* [online]. [cit. 2013-04-01].
<http://xdebug.com/index.php>.

[33] *Debugging Perl* [online]. [cit. 2013-04-01].
<http://docs.activestate.com/komodo/6.0/debugperl.html>.

[34] *Debugging Python* [online]. [cit. 2013-04-01].
<http://docs.activestate.com/komodo/6.0/debugpython.html>.

[35] *Debugging Ruby* [online]. [cit. 2013-04-01].
<http://docs.activestate.com/komodo/6.0/debugruby.html>.

[36] *Debugging Tcl* [online]. [cit. 2013-04-01].
<http://docs.activestate.com/komodo/6.0/debugtcl.html>.

[37] *komodo-debug Node.JS Module* [online]. [cit. 2013-04-01].
<https://nodejsmodules.org/pkg/komodo-debug>.

[38] *Java Compiler Compiler* [online]. [cit. 2013-04-01].
<https://java.net/projects/javacc>.

[39] *Xdebug Remote Debugging Clients* [online]. [cit. 2013-04-01].
<http://xdebug.org/docs/remote#clients>.

A Obsah přiloženého CD a návod k použití

Na disku přiloženém k této práci se nacházejí následující adresáře:

doc Obsahuje soubor ve formátu PDF s textem této práce a citovanou specifikaci protokolu DBGP ve formátech HTML a rST.

examples Obsahuje několik zdrojových souborů v jazyce TL.

tools Obsahuje jednoduchý nástroj v Javě, ke kterému se může připojit běhové prostředí jazyka TL jako k ladicímu jádru, pro ruční testování.

bin Obsahuje soubory:

- **tl.jar** Samostatně použitelný překladač a virtuální stroj jazyka TL.
- **TLProject.vsix** Instalační balíček s rozšířením pro Visual Studio 2012.

src Obsahuje adresáře:

- **TEX** Obsahuje zdrojové soubory k textu této práce.
- **TL** Obsahuje zdrojové kódy syntaktického analyzátoru, překladače a běhového prostředí jazyka TL v jazyce Java a nástroj javacc.
- **VSIX** Obsahuje zdrojové kódy a soubory řešení podpory ladění jazyka TL pro Visual Studio.

A.1 Postup sestavení

Tři podadresáře adresáře src obsahují skripty make.cmd, které zajistí sestavení zdrojových souborů do příslušných výstupních souborů. Skript make.cmd přímo v adresáři src zajistí spuštění jednotlivých skriptů a přesunutí výstupních souborů do správných adresářů.

K sestavení PDF souboru s textem práce je potřeba sada nástrojů LaTeX, konkrétně program pdflatex a několik běžně dostupných balíčků. K sestavení verze na CD byl použit balíček MiKTeX verze 2.9.

K sestavení překladače a virtuálního stroje jsou potřeba programy java, javac a jar verze alespoň 1.6. Tyto programy jsou dodávány ve vývojářském balíčku Java Software Development Kit (JDK). K sestavení verze na CD byl použit balíček od společnosti Oracle verze 1.7.0_17.

K sestavení rozšíření pro Visual Studio je potřeba .NET SDK verze 4 a nainstalované Visual Studio 2012 SDK. Řešení je také připraveno k otevření ve Visual Studiu.

A.2 Postup instalace

Překladač a virtuální stroj jazyka (tl.jar) není potřeba před použitím nijak instalovat. Rozšíření pro Visual Studio je možné nainstalovat spuštěním souboru TLProject.vsix.

Poznámka A.1 Pokud je rozšíření spuštěno v experimentální instanci VS a je k ní připojena hlavní instance VS, dochází při procházení vlastností projektů a při zavírání projektů k výjimkám. Toto není chyba kódu, jde o předepsané chování[15, 16].

B Specifikace jazyka TL

Program je tvořen posloupností příkazů. Příkazy jsou ukončovány středníkem, jinak jsou zapsány ve volném formátu, bílé znaky a konce řádků slouží pouze jako oddělovače a na význam programu nemají vliv. Komentáře jsou omezeny dvěma lomítky a koncem řádku. Klíčová slova jsou rezervovaná. V identifikátorech se rozlišují velká a malá písmena, v klíčových slovech ne.

Názvy proměnných jsou tvořeny identifikátorem. Každá proměnná musí být před použitím deklarována s uvedením svého typu, opakovaná deklarace proměnné téhož názvu je chybou. Po deklaraci nabývá proměnná výchozí hodnoty podle svého typu. Jsou definovány následující typy:

Boolean Nabývá hodnot **true** nebo **false**. Výchozí hodnota: **false**

Integer Obsahuje celé číslo v rozsahu definovaném typem **int** jazyka Java. Typ je v jazyku deklarován klíčovým slovem **int**. Výchozí hodnota: `0`

Float Obsahuje reálné číslo v rozsahu a s přesností definovanými typem float jazyka Java. Výchozí hodnota: `0.0`

String Obsahuje řetězec znaků. Výchozí hodnota: `""`

Jsou definovány následující příkazy:

- Prázdný příkaz.
- Deklarace proměnných uvedeného typu.
- Přiřazení hodnoty proměnné. Proměnná musí být před použitím deklarovaná. Typ výrazu na pravé straně musí být stejný jako typ proměnné.
- Přečtení hodnot ze standardního vstupu a přiřazení do proměnných. Každá čtená hodnota je na samostatném vstupním řádku.
- Výpis hodnot výrazů na standardní výstup. Za hodnotou posledního výrazu je vypsán znak konce řádku.
- Podmíněný blok příkazů, podmínka musí být výraz typu boolean. Část else je nepovinná.
- Podmíněný cyklus, podmínka musí být typu boolean.

V tabulce 1 jsou uvedeny všechny definované operátory. Písmena I, F, B, S vyjadřují v tomto pořadí výrazy typu integer, float, boolean a string. Písmeno T zastupuje výraz jakéhokoli z typů uvedených v definici, ale v jedné definici musí zastupovat vždy stejný typ. Kromě toho lze výrazy typu integer použít (mimo pravé strany ternárního operátoru) ve všech kontextech, kde je vyžadován či povolen výraz typu float. V takovém případě se celočíselná hodnota převede na reálnou.

Popis	Operátory	Signatura
Znaménkové operátory	+ −	$T \to T$ ($T = I, F$)
Aritmetické operátory	+ − ∗ /	$T \oplus T \to T$ ($T = I, F$)
Operátor modulo	%	$I \oplus I \to I$
Operátor sřetězení	.	$S \oplus S \to S$
Relační operátory	== !=	$T \oplus T \to B$ ($T = I, F, S, B$)
Relační operátory	< > <= >=	$T \oplus T \to B$ ($T = I, F$)
Operátor negace	!	$B \to B$
Logické operátory	&& ||	$B \oplus B \to B$
Podmínkový operátor	? :	$B \oplus T \oplus T \to B$ ($T = I, F, S, B$)

Tabulka 1: Operátory jazyka TL

Významy operátorů (kromě operace sřetězení) odpovídají definici jazyka Java. Výrazy mohou obsahovat závorky. Operátory jsou zleva asociativní, priorita operátorů je následující (v pořadí od nejnižší k nejvyšší):

1. podmínkový operátor
2. logický součet
3. logický součin
4. relační operátory
5. aritmetický součet, odečet a sřetězení
6. násobení, dělení a modulo
7. unární operátory

B.1 Gramatika

⟨*program*⟩ ::= ⟨*block*⟩ EOF

⟨*block*⟩ ::= (⟨*statement*⟩)+

⟨*statement*⟩ ::= ‘`;`’
| ⟨*type*⟩ ⟨*identifier*⟩ (‘`,`’ ⟨*identifier*⟩) * ‘`;`’
| ⟨*identifier*⟩ ‘=’ ⟨*expression*⟩ ‘`;`’
| ‘`print`’ ⟨*expression*⟩ (‘`,`’ ⟨*expression*⟩) * ‘`;`’
| ‘`read`’ ⟨*identifier*⟩ (‘`,`’ ⟨*identifier*⟩) * ‘`;`’
| ‘`if`’ ⟨*expression*⟩ ‘`then`’ ⟨*block*⟩ [‘`else`’ ⟨*block*⟩] ‘`end`’ ‘`;`’
| ‘`while`’ ⟨*expression*⟩ ‘`do`’ ⟨*block*⟩ ‘`end`’ ‘`;`’

⟨*expression*⟩ ::= ⟨*conditionalexpr*⟩

⟨*conditionalexpr*⟩ ::= ⟨*logicalorexpr*⟩ ['?' ⟨*expression*⟩ ':' ⟨*conditionalexpr*⟩]

⟨*logicalorexpr*⟩ ::= ⟨*logicalandexpr*⟩ ['||' ⟨*logicalorexpr*⟩]

⟨*logicalandexpr*⟩ ::= ⟨*relationalexpr*⟩ ['&&' ⟨*logicalandexpr*⟩]

⟨*relationalexpr*⟩ ::= ⟨*additiveexpr*⟩ [('==' | '!=' | '<' | '>' | '<=' | '>=') ⟨*additiveexpr*⟩]

⟨*additiveexpr*⟩ ::= ⟨*multiplicativeexpr*⟩ [('+' | '−' | '.') ⟨*additiveexpr*⟩]

⟨*multiplicativeexpr*⟩ ::= ⟨*unaryexpr*⟩ [('*' | '/' | '%') ⟨*multiplicativeexpr*⟩]

⟨*unaryexpr*⟩ ::= ('+' | '−' | '!') ⟨*unaryexpr*⟩
| ⟨*unitexpr*⟩

⟨*unitexpr*⟩ ::= '(' ⟨*expression*⟩ ')'
| ⟨*identifier*⟩
| ⟨*literal*⟩

⟨*type*⟩ ::= 'boolean'
| 'string'
| 'int'
| 'float'

⟨*literal*⟩ ::= STRING_LITERAL
| FLOATING_POINT_LITERAL
| INTEGER_LITERAL
| 'true'
| 'false'

⟨*identifier*⟩ ::= IDENTIFIER

Terminály STRING_LITERAL, FLOATING_POINT_LITERAL, INTEGER_LITERAL a IDENTIFIER následují stejná pravidla jako v jazycích C nebo Java.

B.2 Bytekód

Výstupem implementovaného překladače jazyka je soubor s bytekódem, který je pak vstupem zásobníkového virtuálního stroje. Instrukce bytekódu jsou v přeloženém souboru zapsány pro přehlednost v textové podobě s jednou instrukcí na řádek. Každá instrukce má jeden nebo žádný parametr.

Virtuální stroj má k dispozici paměť s náhodným přístupem adresovanou přirozenými čísly. Stroj si také dokáže zapamatovat speciálně označená místa (identifikovaná přirozeným číslem) v seznamu instrukcí a na požádání na ně přesunout místo vykonávání.

Po svém spuštění stroj načte soubor se seznamem instrukcí do paměti a vyčte z něj verzi bytekódu, návěstí a ladicí informace. Před započetím vykonávání porovná verzi bytekódu s nejvyšší podporovanou verzí. Pokud je verze bytekódu vyšší, stroj předčasně skončí.

Virtuální stroj započne vykonávání na první instrukci a po jejím vykonání se přesune na další. Výjimkou jsou instrukce jmp a fjmp, které mohou změnit místo vykonávání skokem. Jakmile se stroj posune za konec seznamu instrukcí, skončí. Validita instrukcí není při vykonávání nijak kontrolována. Pokud nastane při vykonávání instrukce výjimka, stroj předčasně skončí.

Následuje výčet instrukcí bytekódu s jejich významem a pseudokódem:

add Odebere poslední dvě hodnoty ze zásobníku, sečte je a výsledek uloží na zásobník.
Pseudokód: `right = pop(); left = pop(); result = left + right; push(result); ip++;`

sub Odebere poslední dvě hodnoty ze zásobníku, odečte je a výsledek uloží na zásobník.
Pseudokód: `right = pop(); left = pop(); result = left − right; push(result); ip++;`

mul Odebere poslední dvě hodnoty ze zásobníku, vynásobí je a výsledek uloží na zásobník.
Pseudokód: `right = pop(); left = pop(); result = left * right; push(result); ip++;`

div Odebere poslední dvě hodnoty ze zásobníku, podělí je a výsledek uloží na zásobník.
Pseudokód: `right = pop(); left = pop(); result = left / right; push(result); ip++;`

mod Odebere poslední dvě hodnoty ze zásobníku, podělí je a zbytek po dělení uloží na zásobník.
Pseudokód: `right = pop(); left = pop(); result = left % right; push(result); ip++;`

concat Odebere poslední dvě hodnoty ze zásobníku, sřetězí je a výsledek uloží na zásobník.
Pseudokód: `right = pop(); left = pop(); result = left + right; push(result); ip++;`

uminus Odebere poslední hodnotu ze zásobníku a její opačnou hodnotu uloží na zásobník.
Pseudokód: `val = pop(); result = −val; push(result); ip++;`

and Odebere poslední dvě hodnoty ze zásobníku, provede logický součin a výsledek uloží na zásobník.
Pseudokód: `right = pop(); left = pop(); result = left && right; push(result); ip++;`

or Odebere poslední dvě hodnoty ze zásobníku, provede logický součet a výsledek uloží na zásobník.
Pseudokód: `right = pop(); left = pop(); result = left || right; push(result); ip++;`

gt Odebere poslední dvě hodnoty ze zásobníku, porovná je a výsledek uloží na zásobník.
Pseudokód: `right = pop(); left = pop(); result = left > right; push(result); ip++;`

lt Odebere poslední dvě hodnoty ze zásobníku, porovná je a výsledek uloží na zásobník.
Pseudokód: `right = pop(); left = pop(); result = left < right; push(result); ip++;`

eq Odebere poslední dvě hodnoty ze zásobníku, porovná je a výsledek uloží na zásobník.
Pseudokód: `right = pop(); left = pop(); result = left == right; push(result); ip++;`

not Odebere poslední hodnotu ze zásobníku a její negaci uloží na zásobník.
Pseudokód: `val = pop(); result = !val; push(result); ip++;`

push #value Na zásobník uloží hodnotu uvedenou v parametru. Textovou reprezentaci hodnoty v parametru předchází znak určující její typ (I, B, F, S), přičemž hodnoty typu string jsou navíc uzavřeny do dvojitých uvozovek.
Pseudokód: `val = #value; push(val); ip++;`

load #address Načte hodnotu z paměti určenou adresou uvedenou v parametru a uloží ji na zásobník.
Pseudokód: `val = ram[#address]; push(val); ip++;`

save #address Odebere poslední hodnotu ze zásobníku a uloží ji do paměti na adresu určenou v parametru.
Pseudokód: `val = pop(); ram[#address] = val; ip++;`

label #label Označí toto místo v seznamu instrukcí návěstím uvedeným v parametru. Nezmění stav.
Pseudokód: `labels[#label] = ip; ip++;`

jmp #label Přesune místo vykonávání na místo označené návěstím uvedeným v parametru.
Pseudokód: `ip = labels[#label];`

fjmp #label Odebere poslední hodnotu ze zásobníku, a pokud je rovna **false**, přesune místo vykonávání na místo označené návěstím uvedeným v parametru.
Pseudokód: `val = pop(); if is_false(val) then ip = labels[#label]; else ip++;`

print #count Odebere ze zásobníku tolik hodnot, kolik je uvedeno v parametru, a vytiskne je na standardní výstup. Nakonec vytiskne znak konce řádku.
Pseudokód: `repeat #count times { val = pop(); print(val); } print(EOL); ip++;`

read #type Vypíše na standardní výstup výzvu k zadání hodnoty typu uvedeného v parametru (znaky I, B, F, S). Ze standardního vstupu načte řádek a tento text se pokusí interpretovat jako hodnotu uvedeného typu. Pokud se převedení nezdaří, proces opakuje.
Pseudokód: `repeat until success { print_prompt(#type); val = read(#type); } push(val); ip++;`

var #name Podle pořadí a parametrů těchto instrukcí se tvoří tabulka mapující názvy proměnných na jejich adresy v paměti. Stav stroje se nemění.
Pseudokód: `ip++;`

line #line Podle umístění a parametrů těchto instrukcí je možné mapovat řádky zdrojového kódu na sekvence instrukcí. Stav stroje se nemění.
Pseudokód: `ip++;`

bcver #version Označuje verzi bytekódu. Stav stroje se nemění.
Pseudokód: `ip++;`